修士論文

組込み CPU に搭載するリアルタイム音声加工システムの構築

Construction of Real-time voice processing systems by embedded 16 bit-CPUs

報　告　者

学籍番号：1235128
氏名：宮本　崇功

指　導　教　員

綿森　道夫　准教授

令和 3 年 2 月 12 日

高知工科大学　電子・光システム工学コース

目次

第 1 章　序論

1.1　背景

近年のプロセッサ産業における高集積化や高周波化は著しい。我々が普段何気なく使っている電子機器には、用途に応じた様々な IC が使われているが、プロセッサの量産化や高性能化によって、安価で多機能な製品も多く見受けられるようになってきた。そのような中で、音楽や音声処理などといった「音」を扱う分野においてもその恩恵は図りしれない。

もともと「音」を加工や合成をしようとする際には、必ずフーリエ変換ならびに逆フーリエ変換などのプロセスを経なければ扱えない。しかし、このような変換作業は積和演算を含む多くの計算量を必要とするため、マイコンなどの組込みプロセッサでの処理は、パソコンなどに使われている高周波で動作するような CPU とは異なり、動作周波数が遅いために実装が困難である点が挙げられる。そのためマイコンで「音」をリアルタイムで変換することもまた難しい処理であると言わざるを得ない。

しかし、マイコンにも音声処理に長けた DSP と呼ばれる積和演算を高速に計算する組込み回路を標準搭載したものや、DSP 専用のライブラリ関数などが台頭し始め、その敷居は低くなりつつある。本研究で使用した dsPIC もまたその類の一種である。

ところで dsPIC を用いた信号処理において、過去の書籍や文献などを参照してみても、あらかじめ用意した信号データを EEPROM などの記憶域に保存し、そこから加工したものや、マイクで音を録音し、音声データに書き込んでからの処理、ひいては FIR や IIR などのフィルタ係数を事前に準備してリアルタイムに処理する電子工作の例は数多く散見される。[1-4]

しかしながらフーリエ変換の処理をライブラリなどを駆使し、dsPIC でリアルタイムに処理を行い実装した例はいまだ報告されていない。従って本研究では、DSP を内蔵した dsPIC を使い、実際にフーリエ変換と逆フーリエ変換を行って正しく処理がされるかを検証すると共に、これらを使ったマイコンにおける「音」の信号処理や加工、合成といった処理を実装できる新たな可能性について検討していくことに主眼を置いていく。

電子工作の最大の魅力は、昔とは違い使用する電子部品の入手性の良さや安価でも高性能な部品類が沢山世に出回っており、やろうと思えば誰でもすぐに始められ、かつ自分のイメージで作りたいものを作れる創造性があることだと感じている。今までは LED を光らせたり、ブザーを鳴らしたりするなどの工作を行っていたが、今回は新たに「音」の処理を通じて信号処理の仕組みをより身近に感じるとともに、電池などで小型かつ手軽に実装して動かすことができるような形に実現していきたい。

また dsPIC だけでは処理した音を「聞く」ことはできても、実際に「見る」ことはできない。そこで最近になって現れてきた RaspberryPi のような UNIX 系 OS が動くプロセッサを内蔵した小型のパソコンに接続して、dsPIC 側から加工した信号データを送り Python のような人気言語でグラフ表示が行えるような機構を有したリアルタイムシステムを構築したいと考えた。RaspberryPi には汎用 GPIO が 40 ピン備わっているが、昔から現在の最新の 4B モデルに至るまでデジタル信号でしか扱えないというデメリットがある。

すなわち RsapberryPi をアナログで使用する際には外部の A/D コンバータが必要であり、また

フィルタ回路などの周辺回路を組む必要がある。本研究では、A/D 変換ならびに音声加工や処理は dsPIC 側で行うため、RaspberryPi は UART 通信でデータを送るためだけに使用する。そして数々の Python のグラフ処理の例では、Arduino や他のマイコンと繋げて、リアルタイムに値をプロットして表示させるものは少なからずあるものの、本研究のようにフーリエ変換した結果をリアルタイムに表示した例はない。[13-14,22,25]

Python には FFT 専用のライブラリがあることを確認 [25] しているが、先ほど述べたように RaspberryPi 単体では A/D 変換処理ができないため、入力した音声をアナログフィルタを通して回路を組む必要がある。また dsPIC などのマイコンは消費電力も低く電池などで動かせる半面、RaspberryPi では一種のパソコンと同じで、消費電力も大きいために電池では動かず、モバイルバッテリーでも長時間の運用は適さない。加えてモニターやキーボードなども必要であり、信号処理を主で行う環境を構築するには大掛かりである。その点グラフ表示だけであれば、通信に必要な端子を接続するのみで周辺回路は必要なく、またグラフを見たい時に繋げればいいだけであるため、常時接続しておかなければならないものでもない。

従ってマイコンで信号処理を行う今回のシステムでは、小型で手軽にかつ外部との接続も簡略化できるという大きなメリットを持たせることができる。もちろんできることはマイコンの種類によって限られてくるが、他のモジュールとの通信により信号処理においても色々な言語やライブラリと組み合わせることによって、電子工作の世界が広がると期待される。

1.2 目的

本研究では、信号処理に特化した DSP 内蔵の dsPIC シリーズを使い、主として組込みプロセッサにおけるフーリエ変換と逆フーリエ変換の処理をリアルタイムで実現することと、さらに実際に処理した音を加工し、信号がリアルタイムで変換されていることを確認することを目的とする。

実装する際にはフーリエ変換などの理論はもちろんのこと、回路においてもアナログフィルタの併用が必要であり、またプログラムにおいても DSP 専用の組込み関数やサンプリング周期に合わせた A/D 変換とタイマの割り込みなどの処理も必要である。そこで基礎的な計算やマイコンの構造を逐次学習していきながら実装を進めていき、最終的には、電池などで動かす事が出来る音声処理システムの構築を目指していく。

なお本研究では、あくまでもリアルタイム実装に重きを置いているため、市販されている一般的なオーディオアンプなどに使われるようなノイズを考慮した回路製作やパーツ選びは行っていない。入出力部はアナログフィルタに用いられるバターワース型フィルタを採用し、2 次のフィルタ回路の実装を行った。これらを踏まえて、dsPIC を使った組込みマイコンにおける音声処理の今後の課題や音声合成などの検討を行う。

またフーリエ変換したデータを UART 通信で RaspberryPi 上に転送し、Python を用いて matplotlib のライブラリを動かすことでリアルタイムグラフ表示を行い、FFT 処理した後の信号だけでなく、フィルタなどで加工した信号でも正しくグラフ表示できるかどうかを確認した。またその応答性から、dsPIC でより高速なリアルタイム出力をする場合の検討をしていくことも目的とする。

1.3 概要

本研究では最初は dsPIC30F2012 を用いて、FIR 型および IIR 型のフィルタを構成していく。このフィルタを実装するにあたって、まずは各フィルタの伝達関数を求めて、必要なフィルタ係数を計算しプログラムに組み込んでいく。そしてこのフィルタ係数を変化させることによって、任意の周波数特性を得られることを確認する。その後は FFT による実装を検討し、サンプリング周波数に対して変換処理が間に合うことを確認しながら、サンプル数の可変ならびに窓関数などの処理も加えていく。FFT 処理は先述したようにプロセッサには負荷がかかるため、DSP の専用ライブラリを駆使して実装するとともに、実装しなかった場合との比較も行いながら実験を進めていく。なお信号を出力させるときは、PWM を中心としながらも、外付け D/A 部品なども使っていき、実際にスピーカーにつなげて音の変化を確認した。FIR や IIR のフィルタのように事前にフィルタ係数を求めるのではなく、FFT した後の信号は周波数成分を表すので、直接成分値を変えることによって実際にリアルタイムで入出力の信号に変化が起こるかどうかを、また周波数特性が変化されるかも確認していく。

その後は PIC24 のアーキテクチャを搭載している dsPIC33FJ64GP802 を用いて、内蔵のオーディオ専用 D/A での実験を行い、同様の FFT 処理を実装していく。dsPIC30F2012 とは異なり駆動可能クロック周波数も多少高速になり、メモリの RAM の容量も増えているため、その容量を生かした加工や合成処理も検討していき、また最終作品においては実際に信号を加工できるフィルタ処理を実装していきたい。なお仕様は dsPIC30F シリーズとは違い、PIC24F の仕様を参照 [21] しながら組み込んでいくことになる。

FFT 処理の実装とともに、dsPIC を RaspberryPi に接続してデータの送受信を試みる。こちらは dsPIC33FJ64GP802 と最新の RaspberryPi4 を繋げて、Python によるリアルタイムでのグラフ表示を検討した。OS 環境としては 64bit 対応の Ubuntu の場合は wiringpi などのモジュールや raspi-config などが使えない恐れがあるため、公式の RasbianOS を導入して実験を行っていく。dsPIC で FFT 変換を行うことが確認できれば、その値を RaspberryPi に送り、Python にてグラフ表示を行うことができると考えられ、これらから入出力の波形を dsPIC 側で変化させると Python のグラフも追随して変化するかどうかを調べた。

一連の作業にはデジタル信号処理の知識をはじめ、必然的に信号を入出力するアンチエイリアシングフィルタを構成する必要があることから、オペアンプを使った基本的なアナログ回路の学習も並行して行った。一般化された音声加工や信号処理をまとめた書籍や関連する技術を参考 [5-11,24] にしつつ、リアルタイムの実装実験を行うにあたり、過去に行った 12 石のトランジスタを用いたアンプ回路作製や、節点方程式や伝達関数などの基礎知識が生かされてくると考えている。以上のような内容を踏まえて本研究を進めていく。

第 2 章　これまでの実装例と課題

2.1　dsPIC を用いたフィルタの構築

これまで、dsPIC30F2012 を使った FIR 型および IIR 型のフィルタの実装を行った。これらのフィルタは LTSpice などを使って回路シミュレーションを行い、入出力特性を得る事ができない。代わりに各フィルタのブロック図から伝達関数を求め、数式で表現することができる。そして求めた数式を gnuplot でプログラムすることにより、それらのフィルタ係数に伴う周波数特性を得ることができる。実際に使用したタップ数 127 の FIR フィルタのブロック図を下の図に示す。

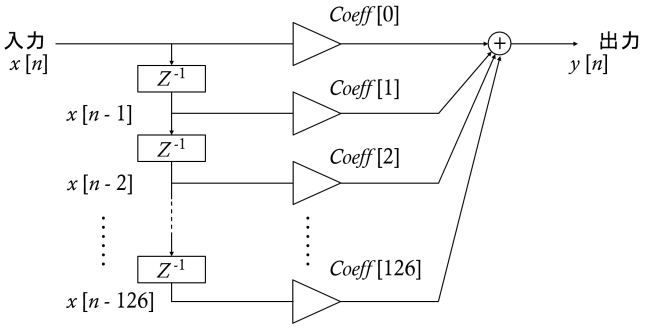

図 1　127 タップの FIR フィルタ

図 1 の C_{oeff} は各遅延器からの信号に掛け合わせる係数であり、この係数を変えることによって、任意のフィルタ特性を持たせることができる。またタップ数を増やすことで計算量は増えるが、より急峻で性能の良い周波数特性を得られる。

2.2　伝達関数とフィルタ係数

図 1 のブロック図から伝達関数を求めてみる。入力信号 $x(n)$ における出力 $y(n)$ の伝達関数 z は式 (1) のように表せる。

$$y(z) = \frac{1}{32768}\left(C_{oeff}[0] + \frac{C_{oeff}[1]}{z} + \frac{C_{oeff}[2]}{z^2} + \cdots + \frac{C_{oeff}[126]}{z^{126}}\right) \tag{1}$$

ここで $y(z)$ を 2^{15} となる 32768 で割る理由として、dsPIC では原則として浮動小数が扱えないため、それぞれの係数を 32768 倍して整数化を行う必要があるためである。次にこの式 (1) を周波数の式に変換するために、次の関係式を用いる。

$$z = e^{sT} \tag{2}$$

$$s = j\omega \tag{3}$$

ここで式 (2) の T はサンプリング周波数 f_s の逆数であり、式 (3) の ω は各周波数を意味する。今回はサンプリング周波数を 28.8kHz として式 (1) に代入すると、

$$y(Z) = \frac{1}{32768}\left(C_{oeff}[0] + C_{oeff}[1]\cdot e^{-\frac{j2\pi f}{28800}} + \cdots + C_{oeff}[126]\cdot e^{-126\frac{j2\pi f}{28800}}\right) \tag{4}$$

の式を得る事ができる。次にこの式の各 C_{oeff} に該当するフィルタ係数を求める。簡易的な求め方として DPSLinks と呼ばれる Windows ツールが書籍 [2] の付録として提供されており、こちらを用いてフィルタ係数を求めた。今回はイコライザのような特性を持たせるべく、図 2 のようにブロックを配置し、各カットオフ周波数を設定して、図 3 のように出力されたフィルタ係数を csv 形式で保存した。

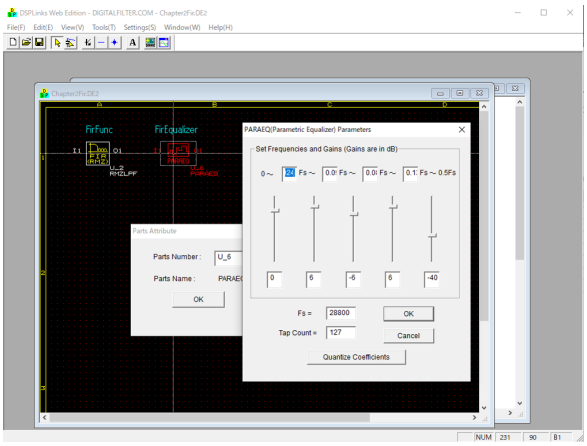

図 2　DSPLinks の設定画面

```
U_6
Parametric Equalizer
Sampling Frequency =   28800.0
fc0 =          0.000 [Hz]
fc1 =        691.200 [Hz]
fc2 =       1497.600 [Hz]
fc3 =       2505.600 [Hz]
fc4 =       4003.200 [Hz]
gain0 =       0.000
gain1 =       6.000
gain2 =      -6.000
gain3 =       6.000
gain4 =     -40.000
Tap Count =127
Quantized by 16 [bits]
             -1
             -1
              0
              0
              0
             -5
            -12
            -16
            -13
             -5
             -1
            -10
            -34
            -59
            -65
            -32
             34
```

図 3　出力されたフィルタ係数

この時に DSPLinks 上では下図のようにフィルタ係数による周波数特性と位相特性のグラフが表示され、作成したイコライザの特性が表示される。

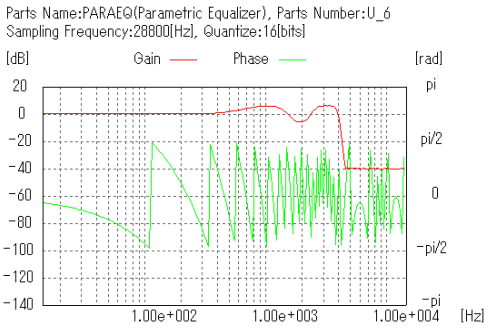

図 4　表示された周波数特性と位相特性のグラフ

ここでイコライザとは周波数帯域を細かく分割し、各帯域でゲインを上下に振ることができる信号処理機能である。今回作成したイコライザは図 4 のように 1kHz 前後のゲインを持ち上げ、2kHz を抑えめに、3kHz 前後を再び上げて、4kHz 以降を大幅に減衰させたものである。

2.3　シミュレーションと実測

出力されたフィルタ係数を用いて、式 (4) の C_{oeff} に数値を代入して計算を行い、gnuplot 上で周波数特性のシミュレーションを行った。そのときの各周波数におけるゲインと位相特性は図 5 のようになった。上の波形が周波数特性であり、下の波形が位相特性のグラフとなっている。先ほどの

DSPLinks 上でも表示された図 4 と同じグラフが表示され、設定したカットオフ周波数の値のとおりに特性が現れていることがシミュレーションからわかる。

次に実際に dsPIC にて FIR イコライザのフィルタ係数をプログラムで書き込んで実測した時の波形を、先程のシミュレーションの波形と合わせて図 6 に示す。

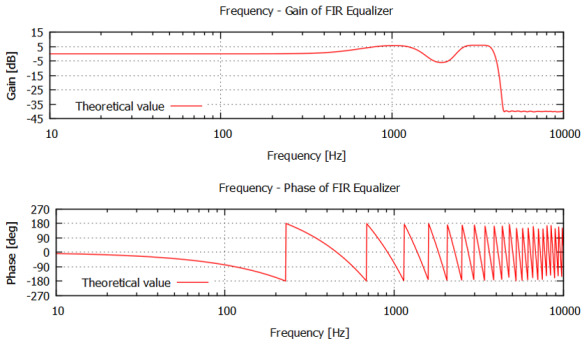

図 5　FIR イコライザのシミュレーション

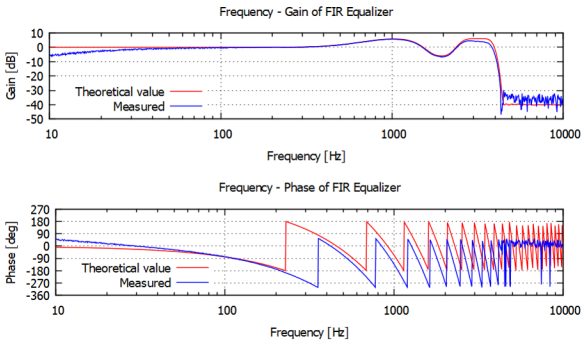

図 6　dsPIC による実測結果

波形結果を見てみると、実測による周波数特性の波形はシミュレーションとほぼ一致していることがわかる。実測の低周波域でのズレは、実測した回路の入力部にあるコンデンサの影響であると考えられる。また位相特性においては波形のズレが生じているが、これは実測の回路に搭載してある前後段のアナログフィルタによる遅延のものだと考えられる。図 5 のシミュレーションにはアナログフィルタの効果は含まれていない。なお回路における入出力部のアナログフィルタのカットオフ周波数は 5kHz となっており、それ以降の周波数は出力されない。

これらの結果から FIR イコライザの特性は、フィルタ係数を DSPLinks 上で生成し、その値をプログラムに書き込むことによって理論値と同じ特性が得られることがわかった。

2.4　課題点

前節のとおり、プログラムで実装した結果はシミュレーションと同じ特性を得られたが、このような FIR 型もしくは IIR 型のフィルタを実装するために、あらかじめ DSPLinks などのツールを用いてフィルタ係数を生成しておく必要がある。そのため、各周波数における特性パターンをテーブルで用意するとプログラムが煩雑になるだけでなく、音声の加工や合成処理における自由度が制限される。そのためフィルタ係数を求めるのではなく、リアルタイムで処理を行う場合には、次項で説明する高速フーリエ変換 (FFT) を行って強度加工を行い、逆高速フーリエ変換 (IFFT) で元の時間信号に戻す方式のプログラムでの実装が望ましいと考えられる。これらの処理は序論でも述べたように、演算に負荷がかかる処理であるが、dsPIC などに搭載されている DSP ライブラリ関数を用いて実装することができる。

第 3 章　フーリエ変換

3.1　離散フーリエ変換の定義

まず最初にフーリエ変換とその逆変換である逆フーリエ変換の定義式を以下に示す。

$$X(f) = \int_{-\infty}^{\infty} x(t)e^{-j2\pi ft}\,dt \quad (-\infty \leq f \leq \infty) \tag{5}$$

$$x(t) = \int_{-\infty}^{\infty} X(f)e^{j2\pi ft}\,df \quad (-\infty \leq t \leq \infty) \tag{6}$$

フーリエ変換とは時刻 $-\infty$ から $+\infty$ までのアナログ信号の周波数特性に用いられる数学的手法であるが、式 (5) と式 (6) における $x(t)$ は時間 t を変数とするアナログ信号、$X(f)$ は周波数 f を変数とする $x(t)$ の周波数特性を表している。

このようなことから、アナログ信号によるフーリエ変換は $x(t)$ および $X(f)$ も無限の連続信号となるが、マイコンを含むすべてのコンピュータでは、連続した信号を扱うことはできない。そのため、次式 (7) と式 (8) の関係に従って t と f をそれぞれ 0 から $N-1$ までの整数 n と k に置き換えた「離散フーリエ変換 (DFT:Discrete Fourier Transform)」を適用することになる。

$$t = nt_s \quad (0 \leq n \leq N-1) \tag{7}$$

$$f = \left(\frac{kf_s}{N}\right) \quad (0 \leq k \leq N-1) \tag{8}$$

ここで t_s は標本化周期、f_s は標本化周波数を表す。DFT は、その逆変換である「逆離散フーリエ変換 (IDFT:Inverse Discrete Fourier Transform)」と合わせて、次のように示される。

$$X(k) = \sum_{n=0}^{N-1} x(n)e^{\frac{-j2\pi kn}{N}} \quad (0 \leq k \leq N-1) \tag{9}$$

$$x(n) = \frac{1}{N}\sum_{n=0}^{N-1} X(k)e^{\frac{j2\pi kn}{N}} \quad (0 \leq n \leq N-1) \tag{10}$$

式 (9) と式 (10) の $x(n)$ は時間 n を変数とする離散的なデジタル信号、$X(k)$ は周波数 k を変数とする $x(n)$ の周波数特性を表す。ここで実際に音データで用いる場合は、$x(n)$ は実数となるが、$X(k)$ は一般的に複素数の値を持つようになる。従って、$X(k)$ は図 7 のような形をとり極座標形式で表すことができ、このときの $X(k)$ の式は次のように示される。

$$X(k) = A(k)e^{j\theta(k)} \tag{11}$$

また図 7 における $X(k)$ の振幅 $A(k)$ と位相 $\theta(k)$ は次式で求めることができる。

$$A(k) = \sqrt{real(X(k))^2 + imag(X(k))^2} \tag{12}$$

$$\theta(k) = \tan^{-1}\left(\frac{imag(X(k))}{real(X(k))}\right) \tag{13}$$

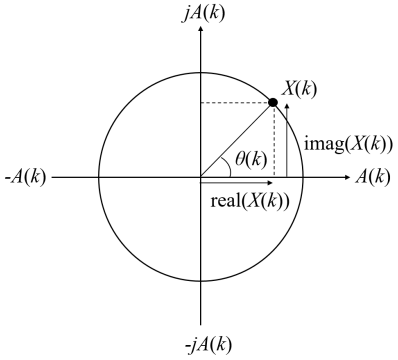

図 7　極座標形式 $X(k)$ の複素数

ここで N サンプルの DFT では N^2 回の乗算および $N(N-1)$ 回の加算を必要とする。すなわち DFT の計算オーダーは N^2 となるため、N が大きくなると計算回数が爆発的に増えてしまうという問題がある。そこで実際に使われているものが「高速フーリエ変換 (FFT:Fast Fourier Transform)」と呼ばれる計算手法である。これは N が 2 のべき乗の値でなければ計算ができないが、代わりに DFT と比べて高速に計算することができる。詳しい計算手順は省略するが、バタフライ計算と呼ばれる方法で計算を行い、これにより N サンプルの FFT は $\log_2 N$ 段で計算され、最終段以外は $\frac{N}{2}$ 回の乗算と N 回の加算、最終段は N 回の加算となるため、FFT の計算は $\frac{N}{2}(\log_2 N - 1)$ 回の乗算と $N \log_2 N$ 回の加算しかなく、N が増えても DFT と比べてかなり高速に処理することができる。

DFT に対する FFT と同様、IDFT を高速に計算できる「逆高速フーリエ変換 (IFFT:Inverse Fast Fourier Transform)」も同じアルゴリズムとなっている。

しかし FFT には計算過程で各要素を並び替えているため、最終的な計算結果である $X(k)$ のインデックスがばらばらになってしまうという問題がある。そのために計算後の $X(k)$ のインデックスを昇順に並び戻すための処理が別途必要となる。これは「ビットリバース (Bit Reverse)」と呼ばれる作業であり、各ビットを逆順にすることにより、ばらばらになっているインデックスを昇順で対応付けることが可能である。なお実際のプログラムにおいてはインデックス並び替えのためのテーブルを作成することが一般的である。[6]

従って、実際に dsPIC でリアルタイムで動作させるためにはプログラムを書き込む際にこの FFT 処理を踏んだあとにビットリバースを行って並び替えをする作業を行う必要があることがわかる。詳しくは次項で述べるが、これらは dsPIC に予め用意されている DSP 専用のライブラリを用いることでプログラムとしては簡単に実装することができるようになっている。

3.2　dsPIC ライブラリによるフーリエ変換の処理

ここから実際に dsPIC に FFT の処理をプログラムで記述していく説明に移る。まず本研究の試作器として用いるプロセッサは dsPIC30F2012 である。動作周波数は内蔵 RC クロック 7.37MHz を 16 倍の PLL とし、1 命令を 4 サイクルで割った 29.48MHz(約 30MHz) である。またここでサンプリング周波数 f_s を 16kHz と設定し、N サンプルを 64 として FFT を試みる。

サンプリング周波数 f_s はタイマー 3 で作成する。動作周波数は 29.48MHz なのでこの逆数をとっ

た 33.92ns が動作クロック時間 T_{cy} であり、これから PR3 レジスタの値を 1843 に 1 を引いた値でスケーリングすることで、16kHz をつくることができる。

使用する A/D コンバータの設定では、タイマー 3 からの割り込みフラグによりデータを取得し自動変換する。そして N サンプル分データを取得した後、FFT 処理を行い PWM で波形を出力させる。次に FFT を実際に処理するために使用する DSP ライブラリの各関数群を説明していく。まずは

```
VectorMultiply( int numElems, fractional* dstV, fractional* srcV1,
                fractional* srcV2 );
```

である。第一引数 numElems はソース・ベクタ内の要素数であり、ここに N サンプルの値が入り今回であれば 64 である。第二引数 dstV は、ディステネーション・ベクタを指すポインタであり、計算された値がこのポインタに返される。なお fractional 型は DSP 用の変数型で、int 型を表している。そして第三引数 srcV1 と第四引数 srcV2 に示されたソース・ベクタの各要素で積をとる。実際に使用するときは、A/D で変換されたデータの要素と窓関数の要素がそれぞれ積をとって計算され、FFT の入力データとして扱われた。

ここで窓関数について説明する。窓関数とは N のサンプル数で整数倍とならない不連続の信号が入力された状態でフーリエ変換を行ったとき、基本周波数以外で本来存在しない周波数成分が広がって現れてしまい、周波数分析の精度を低下させてしまう。この問題を改善するために、入力する信号に釣鐘状の窓関数をかけると、波形の繰り返しに伴う端部の不連続点をあいまいにさせることができ、周波数成分の広がりを抑えることができる。図 8 は窓関数の一種であるハニング窓で、N が 64 の場合の波形を表している。

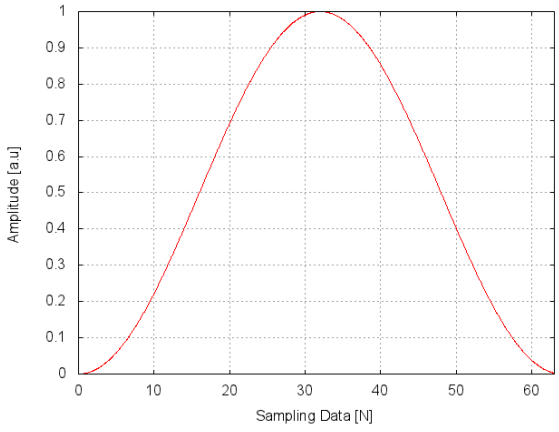

図 8　ハニング窓

このハニング窓の関数を式で表すと次のような形となる。

$$(N \text{ が偶数のとき}) \qquad w(n) = \begin{cases} 0.5 - 0.5\cos\left(\frac{2\pi n}{N}\right) & (0 \le n \le N-1) \\ 0 \quad (otherwise) \end{cases} \tag{14}$$

$$(N \text{ が奇数のとき}) \qquad w(n) = \begin{cases} 0.5 - 0.5\cos\left(\frac{2\pi (n+0.5)}{N}\right) & (0 \le n \le N-1) \\ 0 \quad (otherwise) \end{cases} \tag{15}$$

実際にこの窓関数を実装する場合は、式 (14) と式 (15) などをあらかじめ計算したテーブルを用意しておくか、次のように DSP のライブラリで窓関数を自動で生成する関数を用いて使用する。

```
fractional* HammingInit( int numElems, fractional* window );
```

ここで numElems は N の個数、window は窓関数を示すソース・ベクタポインタである。なお上はハニング窓を生成する関数であるが、この他に Blackman や Kaiser などの関数も用意されている。

次に FFT 処理をする関数のために、引数として用意する変数を準備する。以下のような宣言をして複素変数の作成を行った。なお fftPoints はサンプル数 N を意味する。

```
fractcomplex fftData[ fftPoints ] __attribute__ ((section (".ybss, bss, ymemory"),
                        aligned (fftPoints * 2 * 2)));
```

変換は複素数の形になるため、そのまま C 言語で実装するよりも、DSP の fractcomplex と呼ばれる型を用いた方が楽である。これは任意の変数名の後に.real または.imag を付けるだけで扱うことができる構造体である。そして DSP の関数でフーリエ変換と逆フーリエ変換を行う関数は

```
fractcomplex* FFTComplex( int log2N, fractcomplex* dstCV, fractcomplex* srcCV,
                                fractcomplex* twidFactors, int factPage );
fractcomplex* IFFTComplex( int log2N, fractcomplex* dstCV, fractcomplex* srcCV,
                                fractcomplex* twidFactors, int factPage );
```

である。関数の引数はどちらも同じで、第一引数に N の 2 を底とする複素数 (ソース・ベクタ内の複素要素数) であり、第二引数はディステネーション複素数ベクタを指すポインタ、第三引数はソース複素数ベクタを指すポインタ、第四引数は調整係数のベース・アドレス、第五引数は変換係数のメモリ・ページを表している。

ここで twidFactors について説明する。twidFactors は回転因子と呼ぶもので、複素数の配列からなり、最低でも N の半分の要素が必要である。下図 9 に N が 8 の場合の回転因子の概要を示す。

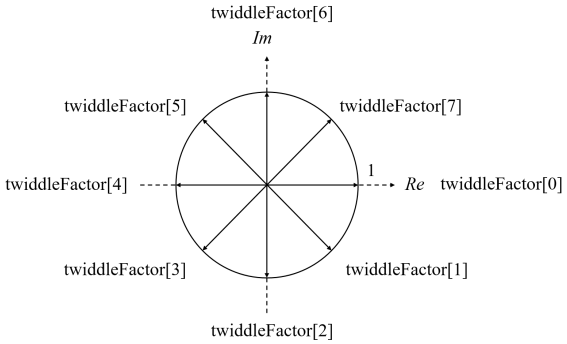

図 9　配列要素における回転因子

$N = 8$ のときの回転因子は、1 周を 8 分割した形となり、実装時は $N = 64$ なので 1 周を 64 分割した形となる。これらの回転因子にその時刻のサンプリングデータをかけ合わせて累積したものがスペクトル波形となる。回転因子は図 9 のとき、twiddleFactor[0] から twiddleFactor[3] までと twiddleFactor[4] から twiddleFactor[7] までが回転因子の向きが真逆なため、前半の半分の係数で容易に残りの係数を作成できる。このことから必要な回転因子の要素数は半分となる。

今回は gnuplot にプログラムを書いてこれらの回転因子の作成を行った。複素数であるため、実数部 Re と虚数部 Im と分けて計算し、結果を 10 進数から 16 進数に変換する作業を行っている。次式は実際に回転因子を生成したときの実部と虚部の計算式である。なお式中の i は for 文で 0 から 32 まで 1 ずつ増やして計算するために用いている。

$$R_e = \cos\left(\frac{\pi i}{\left(\frac{N}{2}\right)}\right) \tag{16}$$

$$I_m = \sin\left(\frac{\pi i}{\left(\frac{N}{2}\right)}\right) \tag{17}$$

プログラムではこの回転因子の値をテーブルで用意したが、下のような関数でも回転因子を作成することができる。

```
fractcomplex* TwidFactorInit( int log2N, fractcomplex* twidFactors, int conjFlag );
```

そして FFTComplex 処理後のビットリバースの処理を行うために、次のような関数を呼び出して、処理をする必要がある。

```
fractcomplex* BitReverseComplex( int log2N, fractcomplex* srcCV );
```

ここでライブラリ関数 FFTComplex と IFFTComplex の仕様を詳しく見てみると、FFTComplex 関数ではビットリバースを行う必要があるが、IFFTComplex 関数では、この関数の中でビットリバースの関数を呼び出してまとめて処理を行ってくれることから、FFTComplex を用いるよりも IFFTComplex 関数をそのまま使用する方が得であることがわかった。もともと逆変換なので、式 (9) と式 (10) を見れば分かるように e の符号が変わるだけなので、回転因子で先ほどの式 (16) と式 (17) の符号を反転させた別の twidFactors を用意して、各 IFFTComplex の twidFactors に代入すればよいことがわかった。

また DSP 関数の中で、FFT 後のスペクトルの大きさを求める SquareMagnitudeCplx と呼ばれる関数があり、これによって式 (12) で表されるような振幅を求めることができる。

```
fractional* SquareMagnitudeCplx( int numelems, fractcomplex* srcV,
                                 fractional* dstV );
```

振幅スペクトルを求める関数はあるが、式 (13) のような位相を計算する関数は用意されていなかった。そのため位相を求めるときは、こちらで各々計算する必要がある。以上が実際に使用した FFT 処理をする上での各関数群である。これらの関数の詳細は microchip 社の DSP ライブラリマニュアルを参照してもらいたい。[16]

3.3　実測結果

これらの関数を用いて作成したプログラムの全容は付録 A にて示すことにするがフーリエ変換した結果をそのまま逆変換して出力するものである。このプログラムを dsPIC に書き込んで、入力周波数を変えたときの出力波形を次に示す。処理としては、入力した信号を変換して何の加工もしないで、そのまま逆変換して PWM で出力させている。

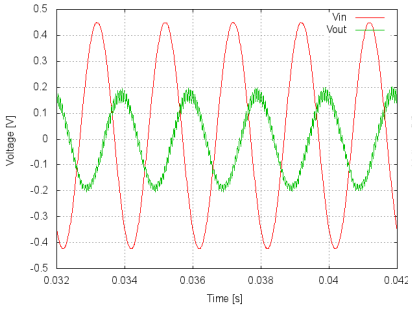

図 10　500Hz の sin 波入力波形

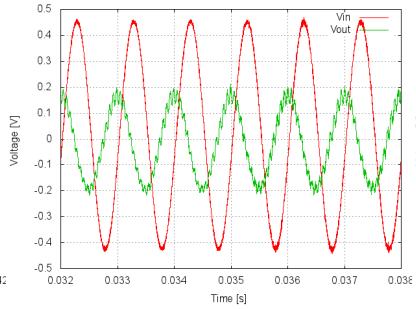

図 11　1kHz の sin 波入力波形

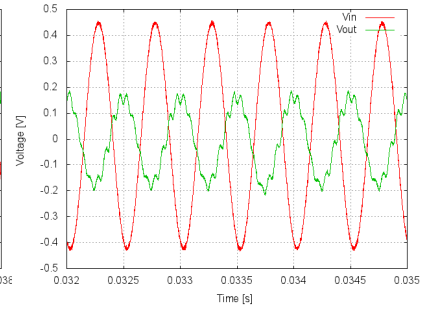

図 12　2kHz の sin 波入力波形

PWM で出力した場合、フィルタを通したあとの波形でも、少しギザギザが残る結果となった。また位相も 90 度ほどズレていることがわかる。この結果から、FFT 処理する前と後では位相が 90 度ほどズレてリアルタイム出力されることがわかる。このとき、出力を PWM ではなく外付けの D/A コンバータの MCP4822 を用いて、同条件下で出力させた場合は次のようになった。

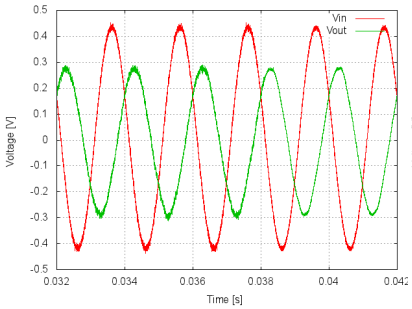

図 13　500Hz での D/A 出力

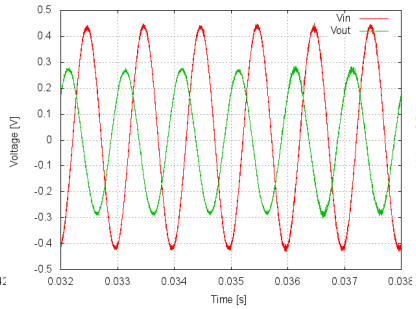

図 14　1kHz での D/A 出力

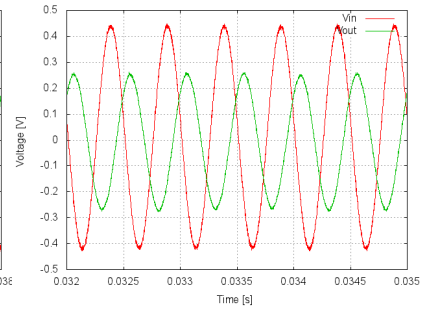

図 15　2kHz での D/A 出力

外付け D/A を使って出力すると、PWM のときと比べて波形がきれいに表示されることがわかる。また位相のズレは同じ 90 度であった。

以上の結果から外付け D/A コンバータを使っても正しく IFFT されて出力できていることがわかり、また PWM と同じで FFT 計算には位相が入出力で 90 度ズレて処理されることがわかった。この処理から次項で述べる音声加工や合成処理についてのプログラム実装を検討していく。

第 4 章　音声加工と応用

4.1　フーリエ変換による音声加工

前述した内容から、dsPIC によるリアルタイムの FFT と IFFT が行えることが確認できた。ここからは入力信号を加工することにより、効果を加えて出力できるかどうかを検討してみる。なお最終的に実装した内容は最終作品の部分で述べるが、ここではその作品で実装していないもので、比較的簡単に実装することができると予想される技術要素を取り上げる。[5]

4.1.1　ディストーション

ディストーションは、信号に歪みをつくり独特の金属音を表現する手法である。ここで図 16 にディストーションを実現するシステムの外観を示す。

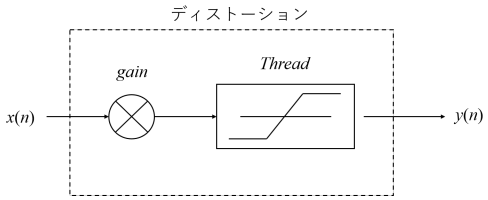

図 16　ディストーションのシステム図

ここで $gain$ は乗算器であり入力信号を増幅させる。直後にしきい値 $Thread$ で決めた任意の値で、それを超える信号と下回る信号はクリップするように増幅した入力の上限下限をカットする方法である。例としては次のようなプログラム構成が考えられる。

```
for( i=0; i<N; i++ )
{
        in_buf[i] = gain * in_buf[i];
        if( in_buf[i] > 2048 )  in_buf[i] = 2048;
        if( in_buf[i] < 1024 )  in_buf[i] = 1024;
}
```

入力信号である in_buf[i] には A/D の 12bit のデータが格納されているが、gain 倍した値で、上限の信号は 2048 とし、下限の信号は 1024 までとして in_buf[i] に格納して個の信号をそのまま出力する。これにより上限下限の信号はクリップし、その間の信号が出力されるような波形が得られる。

4.1.2　ビブラート

ビブラートは、ある周波数を上下に揺らす技法で、歌唱などに用いられるテクニックである。このビブラートは入力信号に周期的な遅延を与えることで実現でき、その遅延は以下のような式になる。

$$\tau = d + d\sin\left(2\pi\frac{r}{F_s}n\right) \tag{18}$$

ここで d は遅延の深さ、r は周波数を意味する。d の値が深いほど周波数方向の変化が大きくなり、r が大きいほど高速な変化が生じる。これを例として記述したプログラムは次のとおりである。

```
        d = 0.002 * Fs;
        r = 5.0;
        for( i=0; i<N; i++ ){
                tau = d + d * sin(2.0 * M_PI * r * l / Fs);
                l = (l+1) % (10*Fs);
                fftData[i].real = fftData[i-tau];
                fftData.imag = 0;
        }
```

変数 l はビブラート用の時刻管理変数で、個の値を変えることで帯域幅を変えることができる。これを dsPIC に搭載することでビブラートが生成できると考えられる。

4.1.3 ゼロ位相変換/逆ゼロ位相変換

ゼロ位相変換は、ゼロ位相領域の信号に変換しこれを逆変換でもとの信号に戻すことで、ノイズ除去に用いることができる手法である。これらは FFT により得られた振幅スペクトルを IFFT をすることで実現できる。振幅スペクトルのみなので、位相スペクトルは使わず 0 であるため、名前の通りゼロ位相変換と呼ぶ。下図にこの変換の流れを示す。

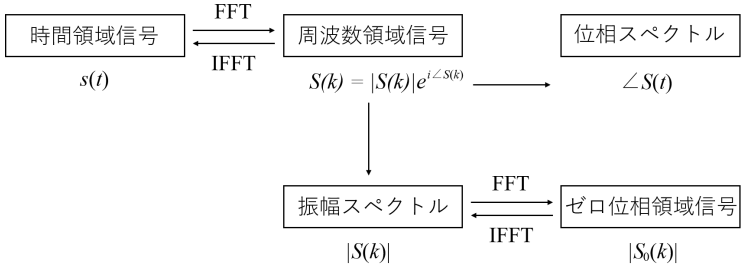

図 17　ゼロ位相変換の流れ

ここで図のように変換された領域をゼロ位相領域、変換された信号をゼロ位相信号と呼ぶ。ゼロ位相信号の FFT は、もとの振幅スペクトルに一致するため、もとの位相スペクトルを保持しておけば、ゼロ位相信号からもとの時間領域信号を得ることができる。

ゼロ位相変換のプログラム例を下記に示す。なお FFT に必要な DSP 関数の引数は省略して記述している。

```
        FFTComplex();
        for( i=0; i<N; i++ ){
        if( fftData[i].real != 0 ){
            phaseData[i] = atan2( fftData.imag[i], fftData[i].real );
        }
        fftData[i].real = pow( powerSpec[i], 1 );
        fftData.imag = 0;
      }
        IFFTComplex();
```

位相を保持するために phaseData[i] の配列を用意し、$-\pi$ から π までのラジアンの値を入れている。その後は、FFT の振幅スペクトルをべき乗 (ここでは 1) して逆変換している。

この逆変換である逆ゼロ位相変換のプログラム例は次のようになる。同様に DSP の関数は簡略して記述している。

```
FFTComplex();
SquareMagnitudeCplx();
for( i=0; i<N; i++ ){
        fftData[i].real = powerSpec[i] * cos(phaseData[i]);
        fftData[i].imag =powerSpec[i] * sin(phaseData[i]);
}
IFFTComplex();
```

これらをプログラムで組み込むことで、理論上は dsPIC でもゼロ位相変換および逆ゼロ位相変換が実装できると考える。

4.2 音声合成処理の検討

今までは音のエフェクトや加工などの処理の一例を列挙したが、ここでは dsPIC を使った音声合成処理を検討する。

4.2.1 ケプストラム変換

ケプストラム変換は、音源の特性と、声道の特性を分離するために用いられる。人間の声は、ノドにある声帯振動により発生する周期的な音源が、声帯から唇、鼻腔までの声道を通過することで生成される。声帯振動は音声の高さを決定し、声道は音声の「あ」や「い」などの特性を決定する。音声をフーリエ変換し、得られた周波数特性 $S(\omega)$ は次式で示される。

$$S(\omega) = H(\omega)G(\omega) \tag{19}$$

ここで $G(\omega)$ は音源信号の周波数特性、$H(\omega)$ は声道フィルタの周波数特性を表し、音声は音源信号が声道フィルタを通過することで得られることがわかる。音声の特性と声道の特性を分離させるために、式 (19) の両辺を絶対値をとり、さらに対数をとると

$$\log|S(\omega)| = \log|H(\omega)||G(\omega)| \tag{20}$$
$$= \log|H(\omega)| + \log|G(\omega)| \tag{21}$$

このように $S(\omega)$ 声道特性と音源特性の和で表すことができる。そして両辺を逆フーリエ変換すると

$$C(t) = IFFT\log|S(\omega)| = IFFT\log|H(\omega)| + IFFT\log|G(\omega)| \tag{22}$$

この流れがケプストラム変換である。$H(\omega)$ は低い周波数成分をもち、$G(\omega)$ は高い周波数成分を持つことが一般的である。これにより原点に近いスペクトラムだけを抽出すれば、声道特性が抽出できる。これにより声道特性は音声の「あ」や「い」を特徴づけるので、音声認識などに利用することができる。一連のケプストラム変換の流れを図 18 に示す。

このケプストラム変換は図のように FFT 及び IFFT をそれぞれ 2 回ずつ行うため、サンプリング周波数や N サンプルの数によっては計算が間に合わない可能性がある。逆を言えば、サンプリング周波数を下げて、N サンプルを増やした場合実装ができるかもしれない。なおプログラムは紙面の都合上省略する。

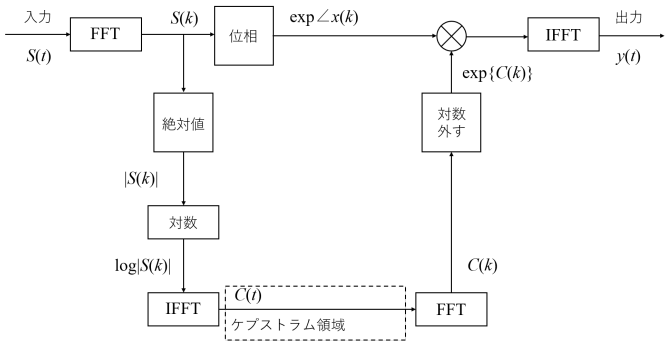

図 18　ケプストラム変換の流れ

4.2.2　線形予測分析と合成

音声の解析技術として線形予測分析、またその分析結果の音声信号を合成することを検討してみる。システムの流れとしては次のようなものである。

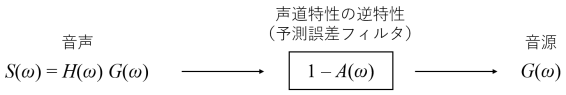

図 19　線形予測分析の流れ

前節で人間が発する音声の仕組みについて概要を述べた。ここで入力する音声から、音源と声道特性を分離させるために線形予測分析が用いられる。図 19 の予測誤差フィルタは FIR フィルタで設計でき、その係数は次式のレビンソン・ダービン・アルゴリズムにより求めることができる。

$$1 - A(z) = 1 - \sum_{m=1}^{M} a_m z^{-m} \tag{23}$$

ここで予測誤差として得られた音源は、声道付近で生じているであろう音となる。この音源にレビンソン・ダービン・アルゴリズムで用いた係数を用いることで、元の音声を合成することができる。

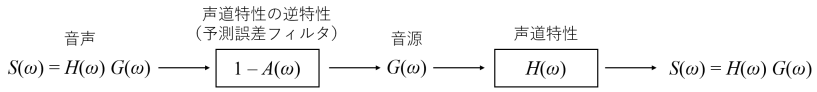

図 20　線形予測分析と合成の流れ

これらを dsPIC に組み込むことで音声の合成処理も行えることができると考える。予測誤差フィルタのレビンソン・ダービン・アルゴリズムは dsPIC で FIR フィルタを構築することで係数を求め、音源を分離し、またリアルタイムで合成する。FFT 処理はしなくてもできそうだが、リアルタイムで抽出することができるかは実装しながら検討する必要があると考える。今回ここで紹介した信号処理ルーチンは、パソコン上の C 言語を利用したシステムや scilab ソフトによるあらかじめ録音したデータに対する処理 [12] に関しては実施してみたが、dsPIC を用いたマイコン上でのリアルタイム処理には課題が多く、実装は完了していない。

第 5 章　Python によるリアルタイムでのグラフ表示

5.1　RaspberryPi と UART 通信

RaspberryPi の汎用 I/O ピンを用いて、UART 接続を行い、dsPIC からデータの送信を試みる。他の通信規格などは、マスター・スレーブを設定しなければならないため、dsPIC でデータを一方的に送りたい都合上、使用すると dsPIC 側はスレーブに設定せざるをえない。これは RaspberryPi がマスターにしかなれないためである。そうすると dsPIC は、外部 D/A に対してはマスターとなり、RaspberryPi との通信はスレーブになるということからプログラムの煩雑化を避けたかったため、対等通信である UART での実装を行った。

今回使用した RaspberryPi4 は現行のモデルでは最新版であり、UART は全部で 5 つ使用できるようになっている。しかし仕様を見てみると、USB で使う UART と汎用 I/O で使う UART に別れた構造になっている。また標準で使われている I/O の UART1 には Bluetooth の設定が行われており、このピンで通信を行うには Bluetooth 設定を OFF にするよう config.txt を書き換える必要がある。今回は UART2 を使用するべく、/boot/config.txt の最後を次のように書き換えて再起動して利用した。なお UART1 を使用したい場合は、#の箇所を外すことで使うことができる。

```
[all]
enable_uart=1
#dtoverlay=disable-bt
dtoverlay=uart2
```

5.2　dsPIC における送信データ処理

まず dsPIC にて UART の設定を行う。レジスタ設定などは付録 B にて示すことにするが、UART 通信を行うボーレート値は 115200 とした。ここで A/D 変換された 12bit の値は、FFT 処理後には fractional 型の型サイズから 16bit になっている。UART で送る場合は一度に 8bit までしか送れないので、2 回に分けて送信する必要がある。送るデータは FFT した配列要素 64 個に対して、標本化定理によりその半分の 32 個のデータを RaspberryPi に送ることとする。よって送信するためのデータを次のような for 文で作成した。

```
for( i=0; i<32; i++ ){
        powerSpec[i] <<= 8;
        valL[i] = (unsigned char)( powerSpec[i] & 0x00fe );
        valH[i] = (unsigned char)( powerSpec[i]  >> 8 );
}
```

powerSpec[i] には FFT したスペクトルの大きさが格納されている。本研究で実測したとき、入力信号に対する powerSpec の大きさは最大値が小さかった (最大が 2.0) ため、これを 8bit 左シフトさせた最大値 512 でを表示させるようにした。また各配列 valL[i] と valH[i] は、16bit の powerSpec をそれぞれ 8bit で分けて、valH[i] に上位 8bit、valL[i] に下位 8bit を入れるようにさせている。ここで valL[i] が 0xFF でなく 0xFE なのは、最大値 0xFF をデータの先頭フラグとして用いるためであり、詳しくは次項の RaspberryPi のグラフ処理で説明する。

実際に FFT したデータを送信すると図 21 のように出力する信号が無信号あるい歪んでしまう瞬間があることがわかった。この結果を検討することによって、毎回 FFT した信号を送る場合は 16kHz のサンプリング周波数に対して、UART のビットレート値 115200bps の送信速度が極めて遅いことが原因となり、プログラム上での送信終了待ち時間が多いことでこのような波形が発生していると考えられた。

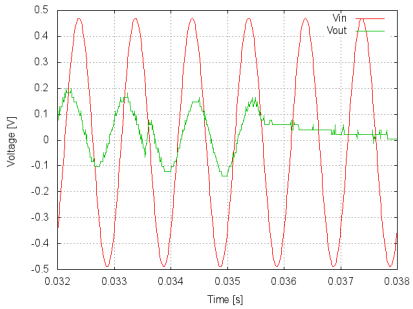

図 21　UART によるデータの連続送信

そこで各処理に要する時間を計算してみた。まずはプログラムでの FFT にかかる時間は次のとおりである。

$$T_f = \frac{64}{16000} = 4\text{ms} \tag{24}$$

FFT はタイマーでサンプリング周波数ごとに割り込みが入り、N に相当するデータが溜まると FFT 変換される。この FFT 変換は main 関数で処理されており、次回の N のデータが貯まるまでに処理しなければならない。そのため式 (24) の T_f はおよそ 4ms かかることがわかる。

続いて UART で送信する処理時間を計算してみる。求める T_u の時間は次のようになった。

$$T_u = \frac{(2 \times 32) + 1 \times 10_{\text{bits}}}{115200_{\text{bps}}} = 5.56...\text{ms} \tag{25}$$

このように UART の送信時間が圧倒的に遅いことがわかる。したがって UART で送信している間は、FFT 処理も止まってしまい、出力信号が無信号となってしまうことがわかった。そのため、毎回 FFT した信号を送るのではなく、500 回のフラグを通した 31.25ms ごとにデータを間引いて処理することにより、この間隔で信号は一瞬だけ 0 になってしまうが、図 22 のように波形を正しく出力されるように改善した。

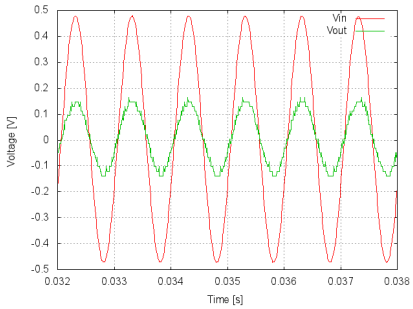

図 22　データを間引きした送信波形

したがって、dsPIC 側の送信間隔の最短速度が 31.25ms となるため、UART のリアルタイム処理ではこれ以上速くすることができない制限が加わった。

5.3 matplotlib によるリアルタイム処理

次に Python を用いてリアルタイムのグラフ表示を試みた。プログラムで最終的に使用したライブラリは以下の 3 つである。[13-16,22]

```
import numpy as np
import wiringpi as pi
from matplotlib import pyplot as plt
```

numpy は、行列や配列といった数値演算に用いるライブラリである。また wiringpi は RaspberryPi の汎用 I/O を UART で使用するために用いた。Python には wiringpi でなく、シリアル通信を行う専用のライブラリが用意されていたが、こちらの wiringPi を使った例が少なかったため、あえてこちらを選択して用いた。そして matplotlib はグラフを表示するために必要なパッケージである。

wiringpi ライブラリで UART を行う方法であるが、PORT を使用するときは次の宣言を行った。

```
serial = pi.serialOpen('/dev/ttyAMA1', 115200)
```

serial は適当な変数で構わないが、PORT を使うときは serialOpen() を用いる。最初の引数には UART2 を使用する場所を指定する。仮に UART1 であれば、ここは ttyAMA0 となる。そして次の引数にはボーレートの値で、dsPIC と合わせて 115200bps とした。

RaspberryPi では dsPIC から送信されたデータを受け取るだけなので、受信処理のみ記述する。wiringpi では以下のように処理できる。

```
if pi.serialDataAvail(serial) :
        c = pi.serialGetchar(serial)
```

serialDataAvail() はデータが 1byte 受信されたときにフラグが立つ関数である。そして serialGetchar() は、バイナリで送られてくるデータを char 型に変換する関数である。シリアル通信では最長 8bit までのデータしか送れないため、dsPIC がデータを 2 回送信したときと同様に RaspberriPi 側でもデータを 2 回受信して、それら結合させる必要がある。

また dsPIC からデータを送るときに、RaspberryPi 側では、先頭のデータから順に配列に格納していきたい。しかし、単に FFT したデータの送受信の処理をしただけでは、延々とデータが送られてくる中でどれが先頭のデータなのかがわからないため、最初に Raspberry 側に送りつけるデータは 0xFF に固定し、これ以外のデータがきても処理しないようにした。なお、dsPIC 側の送るデータは最大値 0xFF から 1 を引いた 0xFE が最大となる。そのため Python で FFT データを受け取る前の先頭データの処理としては、次のような形となる。

```
while True:
        while True:
                if pi.serialDataAvail(serial) :
                        break
        c0 = pi.serialGetchar(serial)
        if  c0 == 255 :
                break
```

2 回目の while 文で dsPIC からデータが受信されるのを待ち、もし最初のデータがきたら変数 c0 に格納して、値が 0xFF であれば全体の while 文を抜けて、以降の処理で FFT のデータを 2 回受信して 16bit に値を結合して処理している。

データを受け取る回数は、dsPIC 側で送信する回数と同じにしなければならないため、32 回の繰り返し処理を行った。dsPIC では 32 回分の FFT したデータを 2 回に分けて送るために合計で 64 回 + 先頭データを送る必要があり、Python でもこのデータをループ間で受信処理しなければならない。それらのデータを格納するために Python で配列の宣言を次のようにした。

```
x = np.array(range(32))
y = np.array(range(32))
```

いずれもグラフに用いる要素で、FFT のデータの個数の 32 個を用意した。また次に FFT のデータを 2 回にわけて受信して処理するプログラムを示す。

```
for i in range(32) :
                        while True:
                                if pi.serialDataAvail(serial) :
                                        break
                        c1 = pi.serialGetchar(serial)

                        while True:
                                if pi.serialDataAvail(serial) :
                                        break
                        c2 = pi.serialGetchar(serial)
                        c = c1 * 256 + c2

                        y[i] = c
```

グラフを書くとき、現在のグラフを表示のみを行う場合は、plt.show() でグラフを表示したあとで、下のような関数を記述する必要がある。

```
plt.gca().clear()
```

この関数がなかった場合、過去のデータがそのままプロットされたままで、値が更新されてもグラフが上書きされて現在のグラフの状態がわからない。そこで、この関数を用いて過去のプロットをいったん消して、現在のデータを新しくプロットしてグラフを更新する必要がある。

今回はグラフをヒストグラムで表示し、そのときのグラフは次のようになった (図 23〜図 25 参照)。なお今回に使用した Python プログラムの全文を付録 C にて示すことにする。

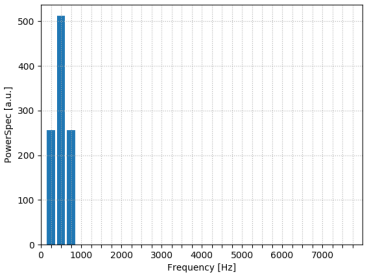

図 23　500Hz でのグラフ

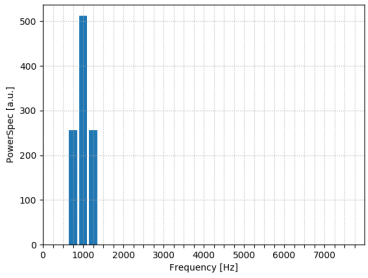

図 24　1kHz でのグラフ

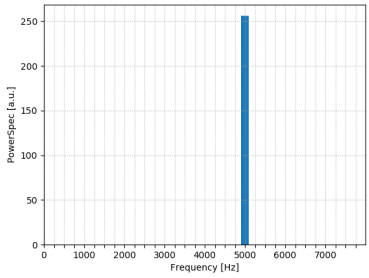

図 25　5kHz でのグラフ

ここでグラフを書く際に for 文を用いて、配列に格納してから表示させる方法を用いている理由について説明する。matplotlib では、x 軸と y 軸を更新しながらリアルタイムで値を表示させることができるように記述することが可能である。しかしそうすると、今回のような予め x 軸の範囲が決まったような処理で、再び 0 に軸を戻すことの処理を実装しにくく、また逐次値をプロットしていく場合では、dsPIC との同期が取りづらいだけでなく、プロットの表示速度も遅かった。そのため、予め必要なデータをすべて受信し配列に格納してグラフを再表示させた方が、リアルタイムとしての速度や応答性が良かったため、こちらを採用した。なお matplotlib には、animation と呼ばれるライブラリ群でグラフを表示するときに、余計な軸や枠など変化しない部分を再表示させず、変化した部分だけを表示させることができ、こちらを使うことによっても実装することが可能かもしれない。

5.4 課題と検討

グラフは正しく表示され、無事 FFT したデータは dsPIC と RaspberryPi 間で送受信され、入力信号を変えてもそれに追随するかのようにグラフの波形が変化することも確認できた。

しかし dsPIC 側の UART 処理で時間がかかることから、サンプル数 N を増やしたり、4 章で述べたような加工や合成などといった、より複雑な処理を行うとなるとさらに時間がかかることが予想される。dsPIC では今回、SPI 通信より外付け D/A と接続しているが、RaspberryPi においても SPI 通信で行った方が、今回の UART で行ったようなデータを間引きするような処理は経なくてもいいかもしれない。またデータを送る dsPIC 側がスレーブ側になり、外部 D/A などのモジュールをつなげている場合はプログラムを複雑にする要因となるため、あまりいい方法ではない。

dsPIC のリアルタイム実装を実現するにあたって、市販のテストボードや解説本などを参考にしたが、SPI で接続した回路例も多く見受けられるため、少なくとも音声信号処理において、他のモジュールと接続する場合は SPI による接続を検討してみる方がより堅実であるかも知れない。ただし外付け D/A コンバータも SPI 通信を利用しているため、dsPIC 型には最低 2 組の SPI モジュール又は ChipSelect による選択が必要となり、ノイズのない通信が可能かどうかはわからない。

第 6 章　入出力部のアンチエイリアシングフィルタ

6.1　アナログフィルタ

6.1.1　バターワース型ローパスフィルタ

音声信号を入力または出力する際には、必ずアンチエイリアシングフィルタが必要となる。本研究で構成した入力側のバターワース型ローパスフィルタについて説明する。

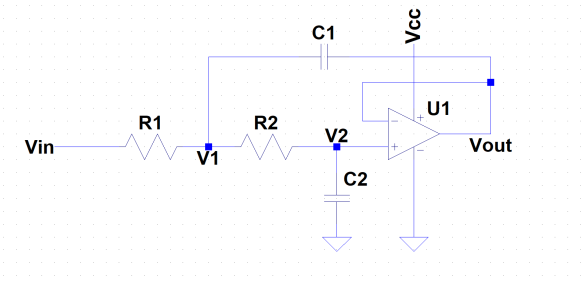

図 26　バターワース型フィルタ

図 26 に示す 1 次のバターワース型フィルタの節点方程式を解く。

$$-\frac{V_{in}}{R_1}+\left(\frac{1}{R_1}+\frac{1}{R_2}+j\omega C_1\right)V_1-\frac{V_2}{R_2}-j\omega C_1 V_{out}=0 \tag{26}$$

$$-\frac{V_1}{R_2}+\left(\frac{1}{R_2}+j\omega C_2\right)V_2=0 \tag{27}$$

$$V_2=V_{out} \tag{28}$$

ここで抵抗 $R_1=R_2$ とすると

$$R=R_1=R_2 \tag{29}$$

$$-\frac{V_{in}}{R}+\left(\frac{2}{R}+j\omega C_1\right)V_1-\frac{V_2}{R}-j\omega C_1 V_{out}=0 \tag{30}$$

$$-\frac{V_1}{R}+\left(\frac{1}{R}+j\omega C_2\right)V_2=0 \tag{31}$$

求めた式 (28) と式 (31) より両辺に R を掛けて整理すると

$$V_1=(1+j\omega C_2 R)V_{out} \tag{32}$$

そして式 (28) と式 (30) を先ほどの式 (32) に代入すると

$$-\frac{V_{in}}{R}+\left(\frac{2}{R}+j\omega C_1\right)(1+j\omega C_2 R)V_{out}-\frac{V_{out}}{R}-j\omega C_1 R V_{out}=0 \tag{33}$$

$$V_{in}=(2+j\omega C_1 R)(1+j\omega C_2 R)V_{out}-V_{out}-j\omega C_1 R V_{out} \tag{34}$$

$$=(2+j\omega C_2 R+j\omega C_1 R-\omega^2 C_1 C_2 R^2)V_{out} \tag{35}$$

$$\frac{V_{out}}{V_{in}} = \frac{1}{1 + 2j\omega C_2 R - \omega^2 C_1 C_2 R^2} \tag{36}$$

$$= \frac{\frac{1}{C_1 C_2 R^2}}{-\omega^2 + j\omega \frac{2}{C_1 R} - \frac{1}{C_1 C_2 R^2}} \tag{37}$$

この式 (37) で導出した解は $H(j\omega)$ と表せる。ここで伝達関数 $H(s)$ を次のようにおく。

$$H(s) = \frac{1}{s^2 + \sqrt{2}s + 1} \tag{38}$$

式 (38) の s を次のようにおいて計算すると、この伝達関数 $H(s)$ の解を求めることができる。

$$s = j\frac{\omega}{\omega_c} \tag{39}$$

$$H\left(j\frac{\omega}{\omega_c}\right) = \frac{\omega_c^2}{-\omega^2 + \sqrt{2}j\omega\omega_c + \omega_c^2} \tag{40}$$

この結果から式 (37) と係数を比較して

$$\frac{2}{C_1 R} = \sqrt{2}\omega_c \tag{41}$$

$$\frac{1}{C_1 C_2 R^2} = \omega_c^2 \tag{42}$$

となり、各 C_1 と C_2 の値を求めることができる。

$$C_1 = \frac{2}{\sqrt{2}\omega_c R} \tag{43}$$

$$C_2 = \frac{\sqrt{2}}{2\omega_c R} \tag{44}$$

式 (43) と式 (44) から $C_1 : C_2 = 2 : 1$ であることがわかる。このことから図 26 のようなフィルタ回路を構成する際に、C_1 と C_2 をちょうど 2 倍となるようなコンデンサの値を決めることによって実装することができる。本研究では、C_1 を 22nF、C_2 を 10nF として回路を構築する。次に抵抗 R の値を決めるが、ここでカットオフ周波数 f_c を 5kHz として式 (43) に代入して計算すると

$$R = \frac{2}{\sqrt{2}\omega_c C_1} \tag{45}$$

$$= \frac{2}{\sqrt{2} \times 2\pi \times 5 \times 10^3 \times 22 \times 10^{-9}} \simeq 2.05....\text{k}\Omega \tag{46}$$

よって約 2kΩ の抵抗をつけることによって図 26 のバターワース特性をもつフィルタ回路が構成できることがわかった。そのためこの回路を入力部に配置して実装を行った。[23]

6.1.2　シミュレーションと実測

実際に LTSpice で作成した 2 次のアナログフィルタを図 27 に示す。前段は入出力 2 倍の反転増幅回路であり、後段が先ほど求めた f_c=5kHz のバターワース回路である。

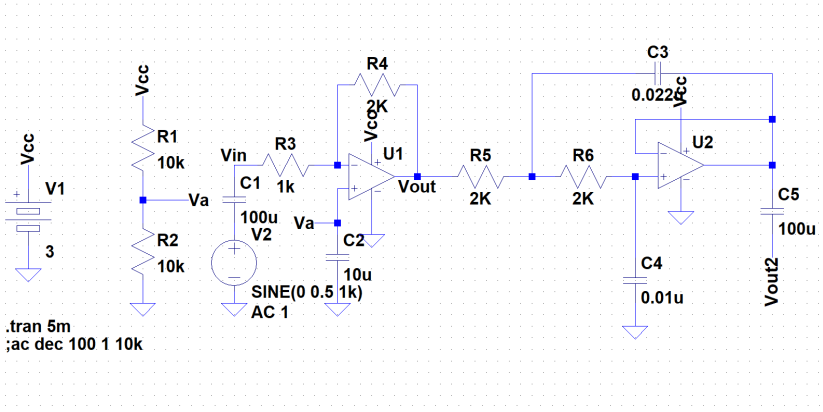

図 27　入力部の 2 次のアナログフィルタ

この回路の周波数特性のシミュレーションは次の図 28 ような結果となり、計算で求めた理論通りの 5kHz で減衰していることがわかる。また振幅 0.5V の 1kHz の sin 波を入力した場合における結果は図 29 となり、2 倍の 1kHz の sin 波が出力されていることがわかる。

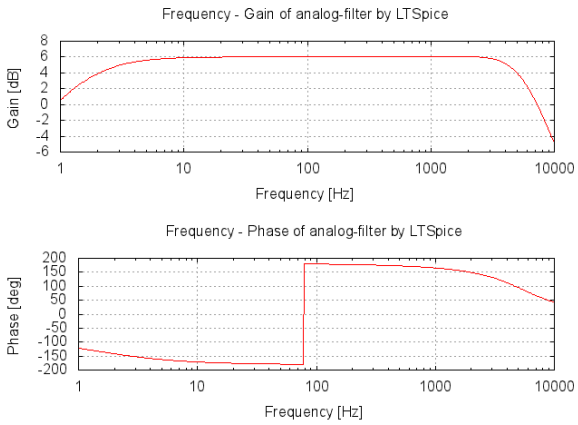

図 28　LTSpice による周波数特性と位相特性

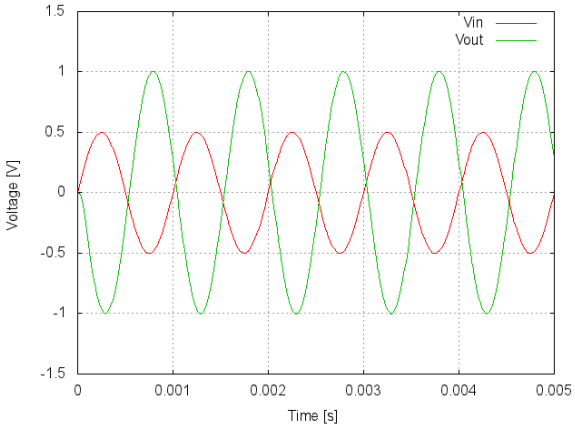

図 29　LTSpice による入出力波形

実際にこのフィルタ回路を組んで実測した周波数特性と位相特性を図 28 と合わせて図 30 に示す。位相特性はシミュレーションとズレているが、周波数特性は 5kHz で減衰していることがわかる。また図 31 の入出力特性はシミュレーションと同じ入力振幅である V_{p-p} を 0.5V にした 1kHz の sin 波を入力したときで、出力信号が入力のおよそ 2 倍で出力されていることがわかる。

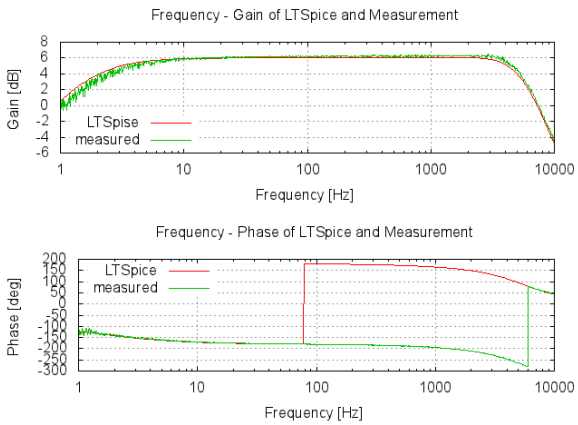

図 30　実測によるゲインと位相特性

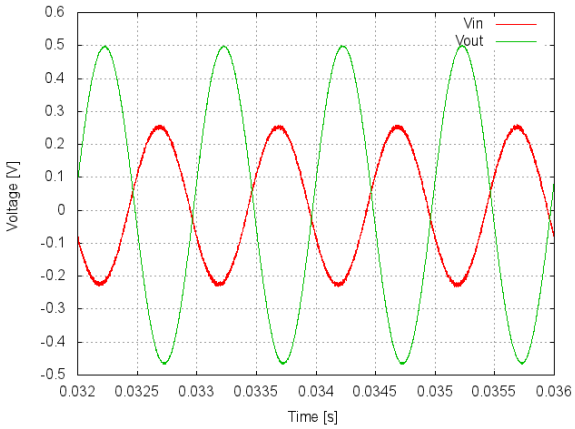

図 31　実測した入出力波形

ここで dsPIC に信号を入力する際は、0 以下の値は扱えないので、電源 3V の半分である 1.5V をオフセットに持たせた信号を入力する。そのため図 27 の初段にあるアンプの正入力に 1.5V を加えている。また周波数特性を測定する際に、直流成分を除くために同図の V_{out} の前にコンデンサを繋いでいるが、dsPIC ではこれをそのまま接続する。

6.2　出力部のフィルタ

6.2.1　フィルタ回路とシミュレーション

前節で入力部のバターワース型フィルタを構成したが、出力部においても同じフィルタを適用した。図 32 に LTSpice で作成したフィルタ回路図を示す。

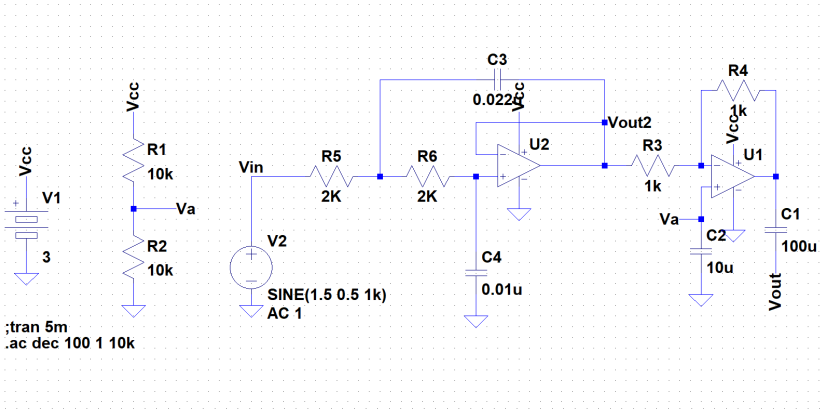

図 32　出力部の 2 次のアナログフィルタ

前段はバターワース回路、後段は入出力等倍のアナログ回路であり、入力部で作成した回路を逆に構成したものとなっている。同様の条件下でこの回路のシミュレーションを行った結果は、周波数特性が図 33、入出力波形が図 34 となり、入力部で構成した周波数特性と同じ特性が得られている。また位相特性は入力部とは違った特性となっていることがわかる。

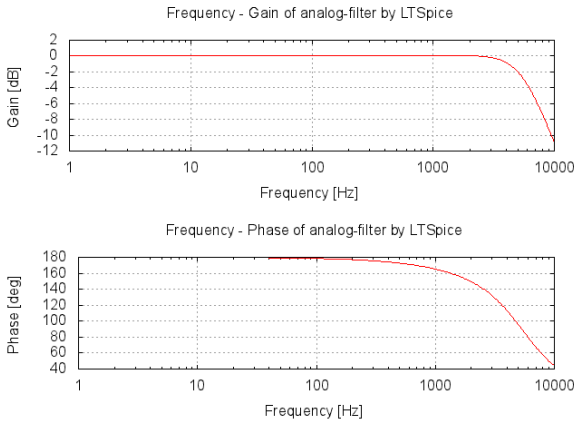

図 33　LTSpice によるゲインと位相特性

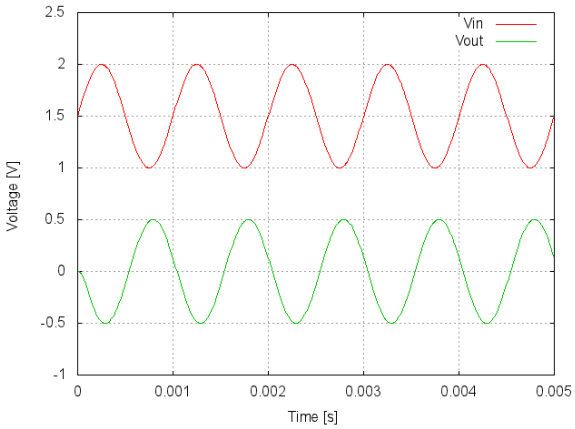

図 34　LTSpice による入出力波形

ここで図 34 における V_{in} は dsPIC からそのまま出力された直後のオフセット 1.5V の信号を含む波形を意味しており、出力信号はコンデンサを通してオフセットを 0V に戻して出力するため、V_{out} のような波形となる。

6.2.2　実測結果

図 32 の回路を実際に作製して測定した周波数特性と入出力特性を下図に示す。

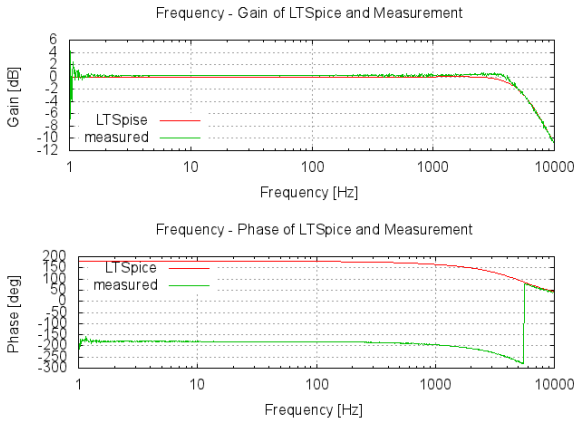

図 35　実測によるゲインと位相特性

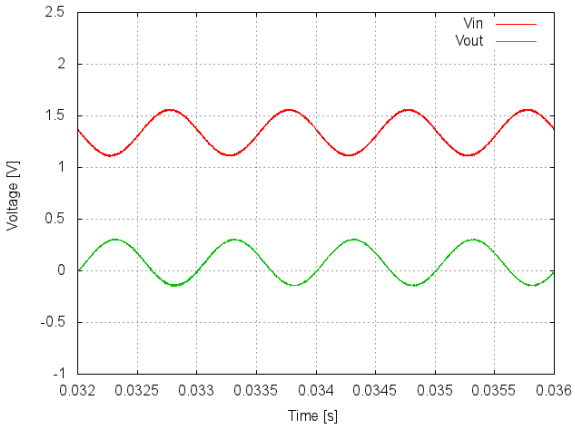

図 36　実測した入出力波形

図 35 をみると、シミュレーション結果と同じ 5kHz で減衰していることがわかる。また位相特性は、シミュレーションとズレていることが分かった。しかしながらズレは 360 度なのでほぼシミュレーション通りの実測結果である。また図 36 の波形をみると、入出力信号が反転した 1 倍の信号が出力されていることがわかる。

6.2.3　オーディオアンプ IC の使用

最終作品では出力部のフィルタ回路にイヤホンジャックを繋ぐとともに、オーディオ専用のアンプ IC を別途繋げて出力させている。この IC を図 32 のバターワースフィルタの後に、コンデンサを通さず、そのままアンプの入力として使用し、スピーカーの音量は可変抵抗で調節することができる。

第 7 章　dsPIC33FJ64GP802 と D/A コンバータ

7.1　内蔵 D/A コンバータによる出力

dsPIC33FJ シリーズは PIC24F シリーズのアーキテクチャを採用した、PIC の中では比較的新しいプロセッサである。これまで本研究の試作段階で用いた dsPIC30F シリーズと主な違いは、使用できるプログラムメモリと RAM の大きさが増えたこと、そしてこの dsPIC33FJ64GP のモデルは内蔵で 16bit のオーディオ用 D/A コンバータを搭載していることであり、魅力的なチップである。試作で用いた dsPIC30F2012 は RAM の容量が 1KB と少なく、FFT 処理を行う上ですぐにメモリオーバーとなってしまったが、新たに実装する dsPIC33FJ64GP802 は RAM が 16KB と格段に増えているため、サンプル数 N を増やしたり、より高度な処理が行うことができると期待される。[18-19,21]

dsPIC33FJ64GP802 の内蔵 D/A コンバータは以下の図 37 の構成となっている。出力されるピンは R チャンネルで RP と RN の 2 ピン、L チャンネルで LP と LN の 2 ピンとなっている。なお RM と LM のピンは出力されないようになっている。

DAC の各チャンネルの出力波形を図 38 に示す。いずれのチャンネルの波形も入力範囲が 0x0000 から 0xFFFF までの信号に対して、下限 VDACL から上限 VDACH なる範囲までの値を出力することがわかる。そして VDACM は中央値であり、RP と LP はポジティブで、RN と LN はネガティブでそれぞれ真逆の波形が出力される。

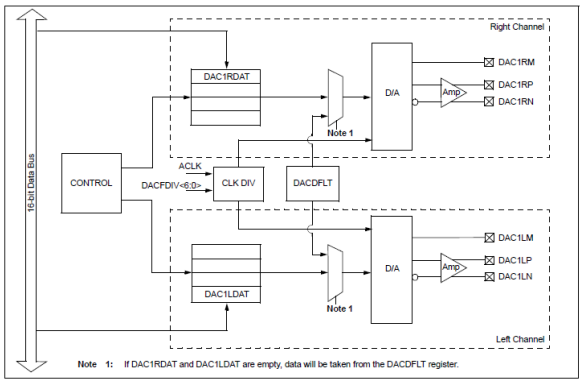

図 37　dsPIC33FJ64GP802 の内蔵 DAC ブロック図

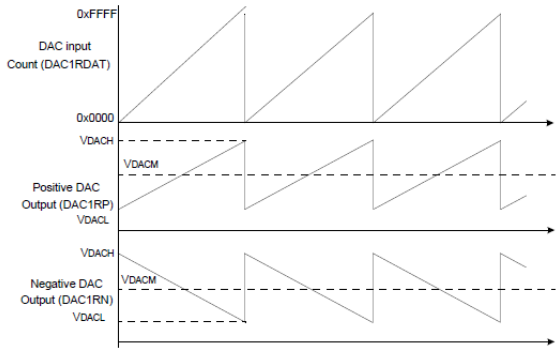

図 38　各チャンネルの出力波形

DAC の変換処理のダイアグラムは次の図 39 のとおりである。両チャンネルにはそれぞれ 4 つのバッファが備わっており、バッファにデータが入ってきたらフィルタを通って $\Sigma - \Delta$ 変調器を通り、再構成フィルタで処理された後、内蔵のアナログフィルタを介してピン出力される。なおバッファのデータが送り終わって、次のデータがバッファに入るまでの間は自動で DFLT レジスタに格納されているデータをフィルタに送っている。

ここで各処理を行うクロックは、dsPIC33F 本体の内蔵クロック構成 (図 40) より後段の PLL 分周器 N2 に入る前の F_{VCO} がクロック源とされる。このクロックの範囲は 100MHz から 200MHz の範囲になるように設定しなければならない。そして使用する DAC のサンプリング周波数によって適

切なクロックレートを分周させて用いる必要がある。そのため今回のサンプリング周波数は 16kHz であるので、256 倍した 4.096MHz が必要な DAC クロックレートとなる。

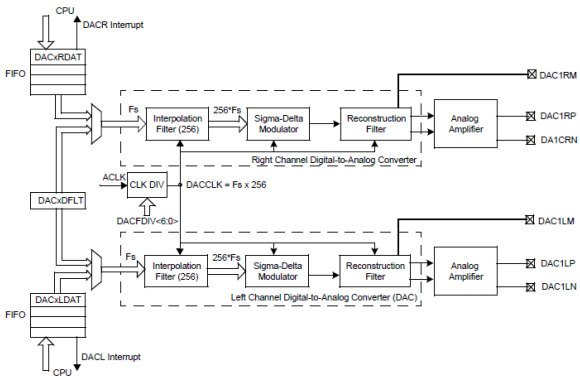

図 39　D/A 変換処理の流れ

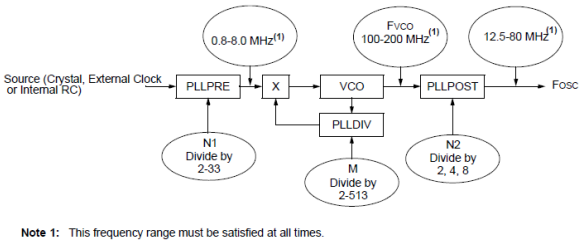

図 40　dsPIC33F のクロック構成

ここで DAC のクロック源の周波数 F_{VCO} は次式より求めることができる。

$$F_{VCO} = \frac{7.3728 \times 10^6 \times 43}{2} = 158,515,200\text{Hz} \tag{47}$$

このとき M の値は 43 として計算した。従って、式 (47) の周波数から 4.096MHz のクロックレートを作るには、F_{VCO} を 39 分周させることで得られることがわかる。そのため、この値を DACFDIV レジスタに書き込むことにより、クロックを生成することができる。

また microchip 社が提供しているの DAC のマニュアルには外付けの出力回路例が記載されている。その回路を図 41 に示す。[17]

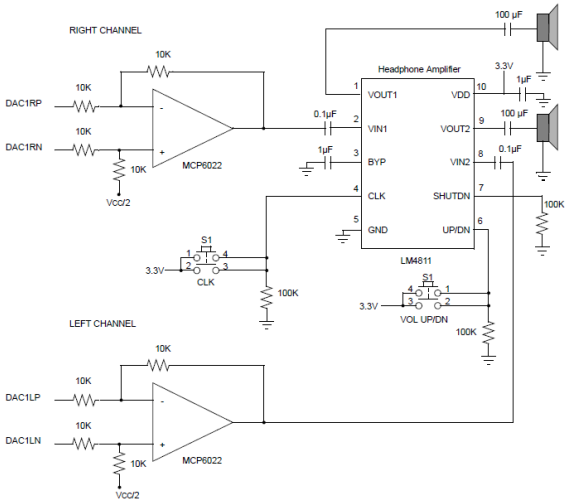

図 41　外付け回路の一例

この例ではオーディオ DAC の差動出力を使用し、差動増幅構成のアンプを用いている。本研究において回路を組み込む際は、図 26 のようなフィルタ回路を前段において処理して行った。また出力にバッファは必要なく、出力信号がそのままピンから取り出すことができることも確認した。

7.2　実際の出力結果

内蔵 DAC で実際に入力信号の周波数を変えてフィルタを通した出力波形を下記に示す。

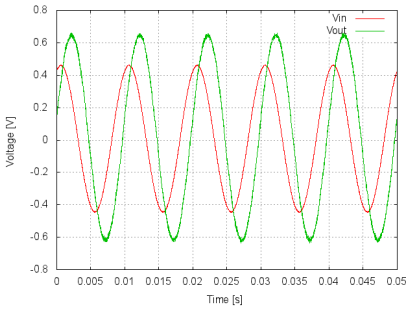

図 42　100Hz の sin 波入力波形

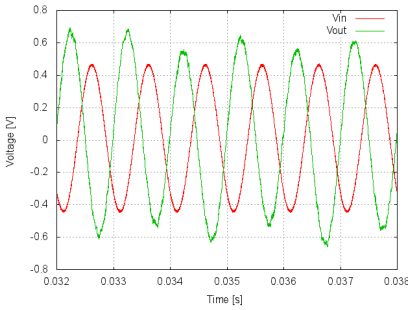

図 43　1kHz の sin 波入力波形

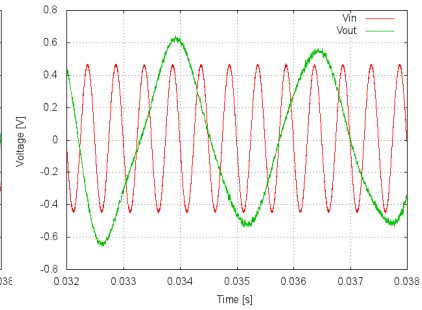

図 44　2kHz の sin 波入力波形

図 42 のような低周波では位相は 90 度ズレているが、波形はきれいに表示されていることがわかる。しかしながら図 43 に示す 1kHz の場合では波形が少し不安定になり、図 44 の 2kHz では周波数すら合わなくなった。これらの原因は不明だが、クロック設定に必要なレジスタの値がまだ不十分の可能性があり、設定を再度見直す必要があると考えた。この状態で周波数特性を測定した結果が図 45 である。

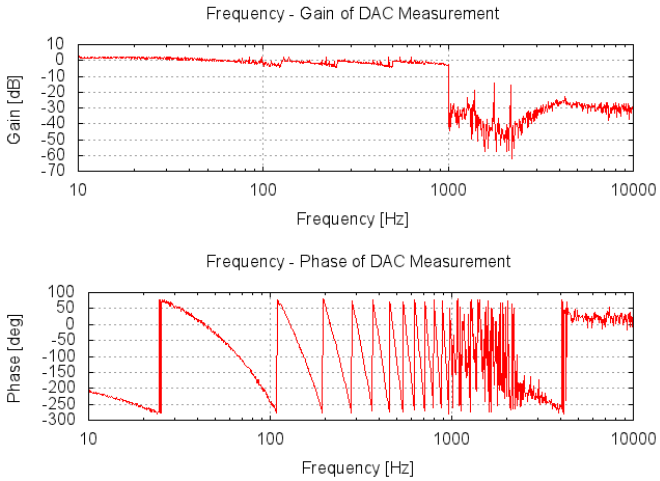

図 45　内蔵 DAC による周波数特性

このように 1kHz 以降から完全に信号がカットされており、正しい周波数特性が得られていないことがわかり、1kHz 以降の信号が全く出力されていないと考えられる。内蔵されている性能のよいオーディオ専用の 16bitDAC を備えているだけに、本機能を使って実験ができないのは真に残念である。

7.3　PWM での出力

dsPIC 内蔵の DAC が使えなかったため、アウトプットコンペアモジュールを使った PWM で出力することも試した。このブロック図は次のとおりである。これをみると、その他の PIC に搭載されているものと同じ構成になっていることがわかる。[20]

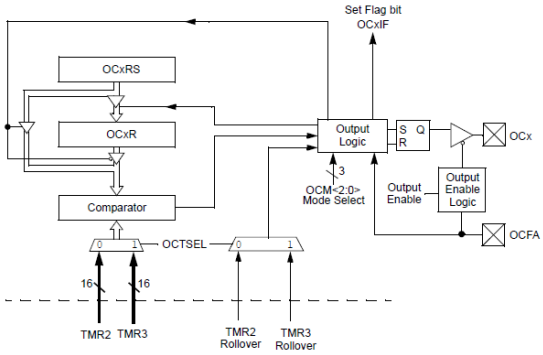

図 46　アウトプットコンペアモジュールのブロック図

タイマー 3 の割り込みをフラグを用いて PWM の処理を行うが、結果として入力した信号をそのまま出力すると、図 47 のように波形が全く出力されなかったため、内蔵 DAC 同様にレジスタ周りの設定やリマップピンの実装配置を見直して出力する必要があると考えられた。

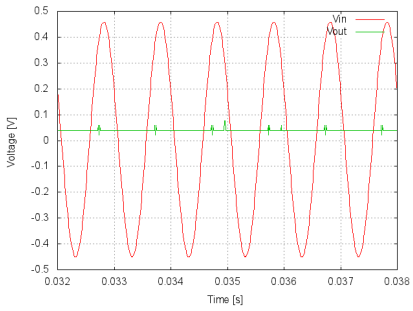

図 47　PWM での出力結果

7.4　外付け D/A コンバータでの出力

外付け D/A を用いて波形の出力を試みた。今回使用した D/A コンバータは microchiop 社の MCP4822 であり SPI で接続するタイプである。dsPIC30F2012 のときに使用したものと同じである。この IC によるプログラムについては付録 D に示すことにする。この D/A コンバータをフィルタ回路に繋いで、周波数を変えて出力すると図 48〜図 50 のようになった。

図をみると波形はどの周波数でもきれいに表示されている。また 100Hz のときの位相は揃っているが、1kHz からずれはじめ、2kHz では位相が 90 度ほどずれていることがわかる。これらから 1kHz 単位で周波数を高くすると位相が少しずつズレることがわかった。またこの外部 DAC の周波数特性は図 51 のような結果となった。

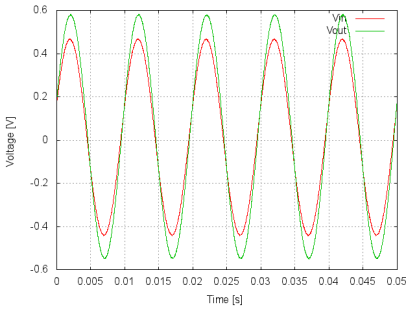

図 48　100Hz の sin 波入力波形

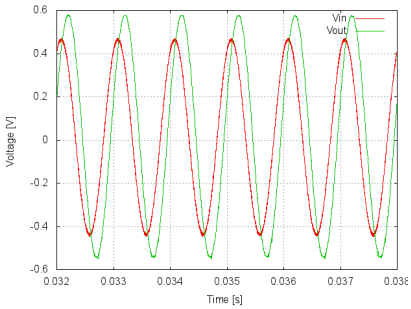

図 49　1kHz の sin 波入力波形

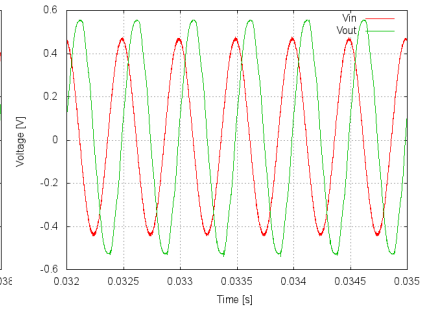

図 50　2kHz の sin 波入力波形

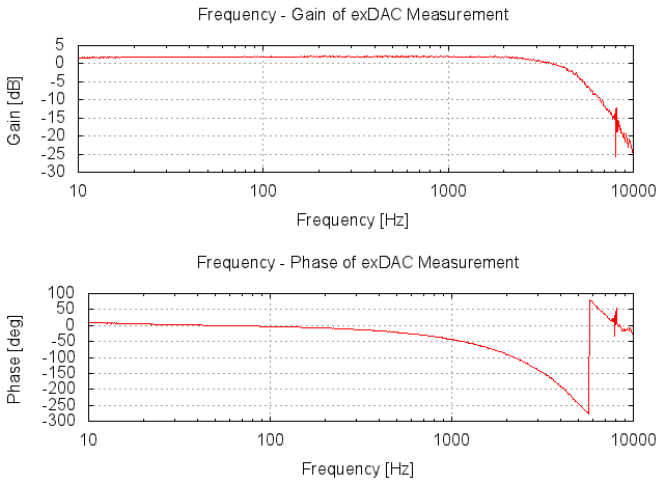

図 51　外部 DAC による周波数特性

図 51 をみると、周波数特性は理論値と同じ 5kHz で減衰していることがわかる。位相は前の図 33 のシミュレーションの時と比べるとズレてはいるものの、図 35 のアナログフィルタのみの場合の位相特性と一致している。

これらを通して、内蔵 DAC と PWM 出力はうまく機能しなかったものの、外部 D/A を用いた信号は出力されることがわかったので、次項で述べる最終作品にはこの IC を用いて FFT 処理を行った。

第 8 章　最終作品とフィルタの構成

8.1　外観と回路図

実際に作製した作品の外観を示す。また回路図は付録 E に掲載する。スピーカーはイヤホンジャックの端子 R に繋がっており、dsPIC で処理を行った信号を出力する。L は THROUGH になっており入力の L チャンネルの信号が出力される。また USB 端子は電源を供給するために用いることが可能である。なお測定用にいくつかのテストピンを備えている。

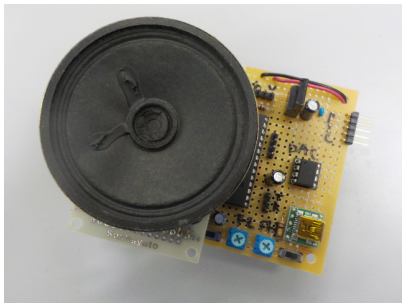

図 52　全体の外観

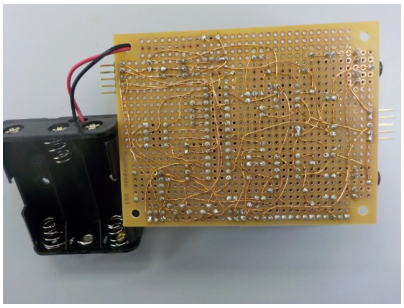

図 53　裏面の配線

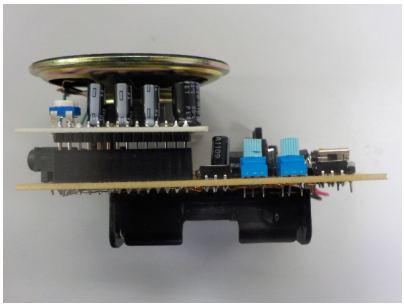

図 54　側面

またスピーカー部の写真は次のとおりである。見えずらいが図 55 のスピーカーの隣にある可変抵抗によって、出力の音量調節を行うことができる。

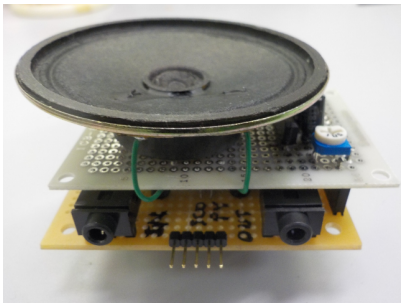

図 55　出力部

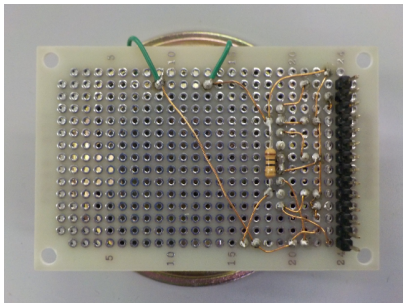

図 56　スピーカーの配線

このシステムに搭載したそれぞれの部品の機能を以下に示す。図 57 の青色の枠で囲まれている部分は入力のアナログフィルタの回路であり、緑色の枠で囲まれた部分は出力用のアナログフィルタの回路である。動作電源は 4.5V(電池) か 5V(USB 入力) であるが、入出力のアナログフィルタに用いている OP アンプ (LMC662) の電源にのみ供給し、レギュレータで 3.3V に降圧してから dsPIC や他の D/A コンバータやスピーカーを鳴らすための IC に電源を供給している。レギュレータが動作する電圧は乾電池 3 本の 4.5V でも駆動する。また別途 USB ジャックで電源 5V を供給することも可能である。

本来は乾電池 2 本の 3V で動作させたかったが、単電源 3V の Rail-to-Rail の OP アンプがなく、

動作も不安定だったため、5V で動作する OP アンプを使用した。また選んだ dsPIC33FJ64GP802 も動作電圧は 3.0V〜3.6V であり、5V では動かないためレギュレータを用いた回路を作製した。なお試作として用いた dsPIC30F2012 は 5V でも動作させることが可能となっている。

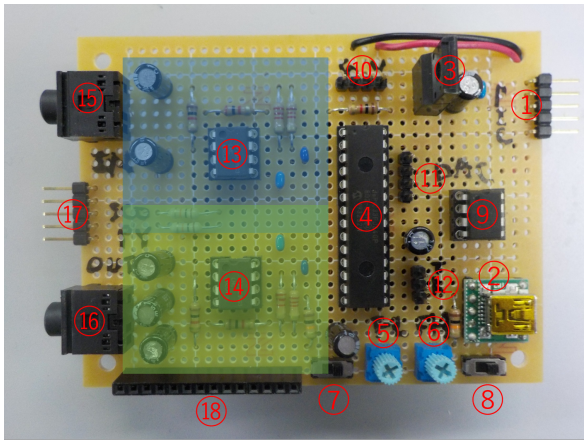

図 57　各機能の詳細

①	PIC 用書き込みピン	②	電源供給用 USB ジャック
③	3.3V 降圧レギュレータ	④	dsPIC33F64GP802
⑤	低域通過フィルタ	⑥	高域通過フィルタ
⑦	SW1	⑧	SW2
⑨	MCP4822(D/A コンバータ)	⑩	外部電源供給用ピン
⑪	内蔵 DAC 出力ピン	⑫	UART 通信用ピン
⑬	入力フィルタ用オペアンプ (LMC662)	⑭	出力フィルタ用オペアンプ (LMC662)
⑮	入力用イヤホンジャック	⑯	出力用イヤホンジャック
⑰	入出力用信号ピン	⑱	スピーカー接続端子

8.2　実装したフィルタ処理

実装したフィルタについて説明する。リアルタイム処理で、FFT した信号から LPF と HPF を行えるようにするものである。図 57 に実装しているスイッチおよび可変抵抗によって信号を切り替えて、周波数域を変えて出力することが可能となっている。

FFT 結果は配列に格納される構造になっているため、各抵抗から入力された値を A/D コンバータを用いて 12bit でデータを受け取った後、次のような処理を行っている。

```
        // HPF
        Freq1 = indata2 * 32 >> 12;
        if ( Freq1 > 31 ) Freq1 = 31;

        // LPF
        Freq2 = indata3 * 32 >> 12;
        if ( Freq2 > 31 ) Freq2 = 31;
```

indata2、indata3 は A/D 変換された値が入っており、これを 32 倍して 12 ビット右シフトさせている。この作業により、データは N の半分の 32 個であるため、A/D の範囲を 0 から 31 までの値で変化させるようにさせている。そして変数 Freq1 と Freq2 にセットされた値で該当する配列要素を 0 にして周波数をカット、あるいは通過させるようにしている。以下が実際のフィルタ処理であるが、ここにスイッチのフラグ処理をいれて記述している。詳細なプログラムは付録 F に示すことにする。

```
        for( i=0; i<Freq1; i++ )
        {
                fftData[i].real = fftData[i].imag = 0;
                fftData[63-i].real = fftData[63-i].imag = 0;
        }

        for( i=Freq2; i<32; i++ )
        {
                fftData[i].real = fftData[i].imag = 0;
                fftData[63-i].real = fftData[63-i].imag = 0;
        }
```

ここで各配列要素に対応する周波数について説明する。このプログラムのサンプリング周波数 f_s は 16kHz であり、N ポイントの FFT を行っているため、各配列と周波数の関係は次のようになる。

$$Data[i] = \frac{f_s \times i}{N} \tag{48}$$

この式 (48) から配列の要素が 1 増える度に、周波数では 250Hz ずつ増加していくことがわかる。なお N サンプルの数を増やすとより周波数を細かくでき、反対に減らすと周波数は大きくなる。今回は 250Hz なのでそれ以下の周波数の信号は残念ながら処理できない。また出力部のフィルタは 5kHz のカットオフなので、範囲としては 250Hz から 5kHz までの信号を処理することができる。

8.3 実測とグラフ

プログラムを組み込んだ LPF での実測結果を下図に示す。

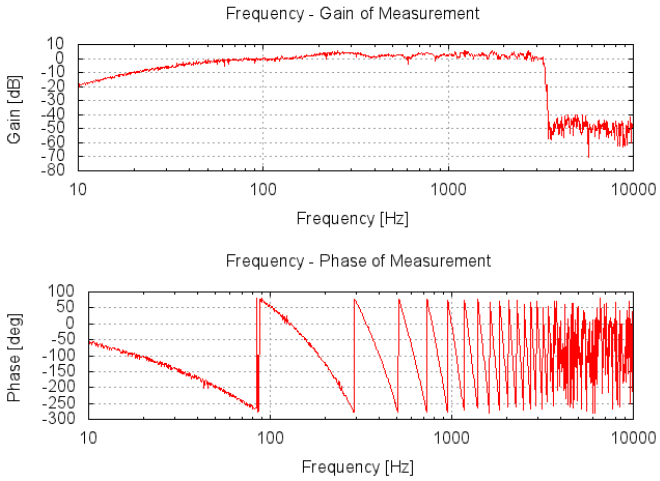

図 58　FFT 処理後の LPF 特性

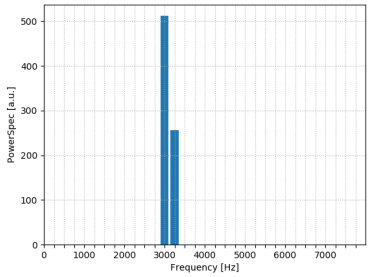

図 59　入力信号 3kHz の場合

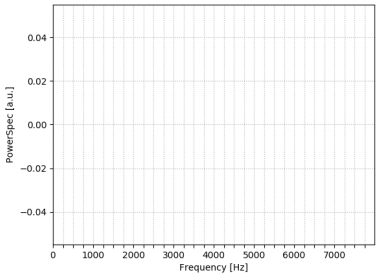

図 60　入力信号 4kHz の場合

図 58 でカットオフ周波数が 3kHz を過ぎたあたりなので、Python のグラフでは 3kHz の信号が出力されているが、入力が 4kHz ではグラフに表示されていない。よって Python 側でも正しく LPF の特性が反映されていることがわかる。次に HPF の場合での実測結果は次のようになった。

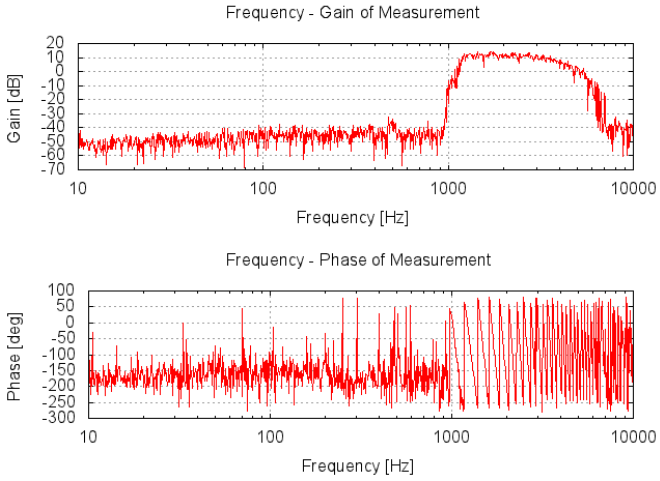

図 61　FFT 処理後の HPF 特性

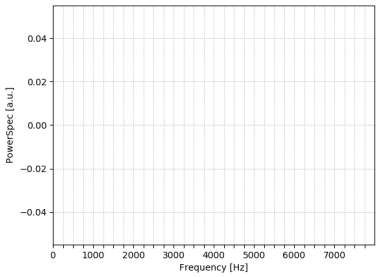

図 62　入力信号 500Hz の場合

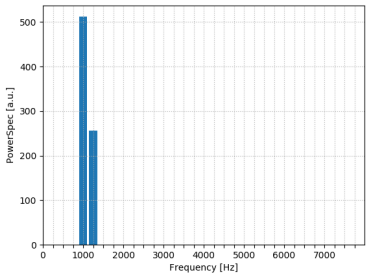

図 63　入力信号 1kHz の場合

図 61 でカットオフ周波数が 1kHz を過ぎたあたりなので、Python のグラフでは 500Hz のときに波形が出ず、入力が 1kHz のときに表示された。このことから Python 側でも正しく HPF の特性も反映されていることがわかる。以上からフィルタ特性が処理され、グラフ化を行うことができた。

第 9 章　結論

本研究より、当初の目的であった組込み CPU におけるリアルタイム処理の実装は行うことができた。今回は信号の加工や高度な合成処理は検討するのみで実装することはできなかったが、LPF や HPF などのフィルタを実装するだけでも、Python のグラフ変化だけでなく、スピーカーを通して耳で音の変化を捉えることができた。IC として dsPIC33FJ64GP802 を用いたが、当初予定していた内蔵の 16bitD/A は使うことができずに、外付けの D/A に頼った処理を行なったが、SPI による通信でもリアルタイム処理に大きな遅延を感じるようなことはなく、FFT 変換処理ができることがわかった。また RaspberryPi における UART であるが、本文でも述べたように普段使っている通信ながら、FFT などをリアルタイムで実装する際に求められる高速な計算速度と比べると、出力に影響を及ぼすだけでなく通信の遅さが伺えた。これはクロックが高速な IC を使っても、ボーレートの値は固定なので、データを送る通信速度は向上できない。従って外部との接続による共同処理においてリアルタイム処理をより厳密に追求するならば、SPI などを採用することが望ましいと考えた。

しかしながら dsPIC と RaspberryPi を繋ぎ、Python でリアルタイムに FFT 処理の結果をグラフを表示することができた。これは dsPIC から RaspberryPi に送られてくるデータを間引いても、リアルタイムに変化するグラフの表示にはあまり問題がないことによっている。入力が変化したときに、値が反映されてグラフに表示される応答性は数秒程度に設定して、dsPIC からのデータを間引きして送っているにもかかわらずリアルタイムに近い動作処理ができているように見えることがわかった。最終作品で搭載したスピーカーで音の変化を聞くこともできるが、Python にあるライブラリを使って視覚的にグラフで音の周波数成分を理解することができ、また今回は FFT のパワースペクトルを表示したが、位相や周波数特性などの必要なデータも加工して送ることで、それらのグラフ化を行うなどの応用もきくことができると考えられる。

また本研究では、回路の実装化にも取り組んだ。RaspberryPi などの大きな電力を必要とする物と比べて、PIC のようなものは小型で電池で動作できるため、一連のフィルタ処理や FFT などの信号処理が電池で動作できることも感慨深い。しかし RaspberryPi は、中の IC が優秀で PIC よりも高速なため本研究のような FFT 処理もスムーズに行うことができる可能性もある。もちろん RaspberryPi には A/D コンバータや D/A コンバータがないので結局外付け IC が必要であるが。

また Python でも FFT できるライブラリやグラフを処理できるライブラリや演算が整っているので、本研究でのリアルタイム処理を Python だけで実現することができるかもしれない。いずれにせよアナログフィルタや A/D のような周辺回路は必要になってくるが、近年の機械学習や DeepLearning などの処理においても、この信号処理分野の知識が役立つに相違ない。本研究により DSP 並びに FFT に関連する処理を学習しながら、電子工作でリアルタイム性と音声処理を実装することができるとわかり、今後の電子工作の幅が広がったと感じている。

今回の様な組込みプロセッサにリアルタイムでフーリエ変換と逆フーリエ変換を組み込む研究は、組み込んだ後に実測によって正しく信号が復元されていれば成功といえる。音声加工としてボリュームを可変させて周波数特性をリアルタイムに変化させた時も、RaspberryPi を通して表示されたグラフが実測の周波数特性が正しいことを想像される様なグラフであれば加工も成功しているといってよい。実際の周波数特性は入力信号の周波数を連続的に可変しながら特性を得るので、これを

RaspberryPi で表示させたグラフと一致することを示すことはできないが、カットオフ周波数前後の周波数の波形を与えれば表示されたり消えたりすることから実装が成功しているといえるであろう。限られた紙面の修士論文においては今回作成したプログラム全文を載せることができず、回路図共々付録に回さざるを得なかった。しかしながら今回のプログラムの実装を dsPIC で行った例は見たことがなく、この論文を読んで興味を持った人が実際に回路を作ってプログラムを入力して試してみることができる様にすることも大切な事であると考え、あえて掲載することにした。

謝辞

この度の卒業研究ならびに修士論文の作成にあたり、始終ご助成を賜りました高知工科大学大学院システム工学群の綿森道夫准教授に心から感謝を申し上げます。本研究は、綿森道夫准教授の丁寧かつ熱心なご指導とご助言によって成り立っています。

参考文献

[1] 後閑哲也，” 周辺モジュールの使い方，” PIC ではじめるアナログ回路，藤澤奈緒美（編），pp.319-323，技術評論社，東京，2014.

[2] 岩田利王，dsPIC 基板で始めるディジタル信号処理，熊谷秀幸（編），CQ 出版社，東京，2009.

[3] 後閑哲也，電子制御・信号処理のための dsPIC 活用ガイドブック，藤澤奈緒美（編），技術評論社，東京，2006.

[4]「dsPIC アプリケーションマニュアル マイクロチップテクノロジー社」: https://www.scribd.com/document/336090713/Memoryskug-Ds-22087

[5] 川村新，” 体感！音声信号処理，” Interface 2016 第 6 号，上村剛士（編），pp.60-90，CQ 出版社，東京，2016.

[6] 青木直史，” 音を見る，” C 言語ではじめる音のプログラミング―サウンドエフェクトの信号処理，pp.21-54，オーム社，東京，2008.

[7] 小坂直敏，サウンドエフェクトのプログラミング ― C による音の加工と音源合成 ―，オーム社，東京，2012.

[8] 君島武志，パソコンで「音」を処理する，工学社，東京，2016.

[9] 松下耕二郎，信号処理のためのプログラミング入門，原田崇靖（編），技術評論社，東京，2009.

[10] 岩田利王，実践ディジタル・フィルタ設計入門，CQ 出版社，東京，2004.

[11] 佐藤幸男，信号処理入門 (改訂 2 版)，オーム社，東京，1999.

[12] 大川善邦，波形解析のための数値計算ソフト Scilab 入門，CQ 出版社，東京，2013.

[13] 後閑哲也，” ラズパイの GPIO の使い方，” PIC と楽しむ Raspberry Pi 活用ガイドブック，藤澤奈緒美（編），pp.89-96，技術評論社，東京，2017.

[14] 福田和宏，”Raspberry Pi のインターフェースと入出力、” 電子部品ごとの制御を学べる！Raspberry Pi 電子工作 実践講座 改訂第 2 版，久保田賢二（編），pp.61-64，ソーテック社，東京，2017.

[15] 柴田淳，みんなの Python 第 4 版，SB クリエイティブ社，東京，2017.

[16] 斎藤和邦，NumPy&SciPy 数値計算実装ハンドブック，秀和システム社，東京，2019.

[17]「dsPIC アプリケーションマニュアル マイクロチップテクノロジー社」: https://www.microchip.com/stellent/groups/devtools_sg/documents/devicedoc/jp019851.pdf

[18]「dsPIC アプリケーションマニュアル マイクロチップテクノロジー社」: http://ww1.microchip.com/downloads/jp/DeviceDoc/70211B_JP.pdf

[19]「dsPIC アプリケーションマニュアル マイクロチップテクノロジー社」: https://akizukidenshi.com/download/ds/microchip/dsPIC33FJ64GP802.pdf

[20]「dsPIC アプリケーションマニュアル マイクロチップテクノロジー社」: http://ww1.microchip.com/downloads/jp/DeviceDoc/70046B_JP.pdf

[21] 後閑哲也，C 言語ではじめる PIC24F 活用ガイドブック，藤澤奈緒美（編），技術評論社，東京，2007.

[22] チーム・カルポ，Matplotlib&Seaborn 実装ハンドブック，秀和システム社，東京，2018.

[23] 久保格致，” 回路の設計，” MicroCap5/CQ を用いたアクティブフィルターの設計と製作，pp.16-18，高知工科大学 2000 年度学士卒業論文，
https://www.kochi-tech.ac.jp/library/internal/ron/pdf/2000/03/42/1010279.pdf，平成 13 年 2 月

[24] 川村新，尾知博，” 音声&画像処理の常識，” ディジタル・デザイン・テクノロジ 2010 NO6，西野直樹（編），pp.20-63，CQ 出版社，東京，2010.

[25] 辰岡鉄郎，”Python で信号処理 — 時系列データ 解析編 —，” Interface 2021 3 月号，pp.19-33，CQ 出版社，東京，2021.

Appendix

付録 A　dsPIC30F2012 FFT プログラム (フィルタなし)

以下に試作で用いた dsPIC30F2012 の FFT 処理プログラムを記載する。動作としては FFT した後に IFFT をしているため、入力信号と同じ波形が出力される。

```
//*****************************************************
// 7.37MHz Internal RC oscillator, 16x PLL enabled
// Fcy=7.37MHzx16/4=29.48MHz, Tcy=33.92ns
//*****************************************************

#include <p30f2012.h>
#define FCY 29480000L
#include <libpic30.h>
#include <dsp.h>
#include <stdio.h>
#include <math.h>

#define powerOf2                6
#define fftPoints       (1<<powerOf2) // N=64, Fs=16kHz, Fs/N=250Hz
#define fftPoints2      fftPoints/2
#define fftMask         fftPoints - 1;
#define Fs      16000

// configuration
_FWDT(WDT_OFF);
_FGS(CODE_PROT_OFF);
_FOSC(CSW_FSCM_OFF & FRC_PLL16);
_FBORPOR(PBOR_OFF & PWRT_64 & MCLR_EN);

// twiddleFactors
const fractcomplex twiddleFactors[] __attribute__ ((space(auto_psv),
aligned (fftPoints * 2))) = {
        0x7FFF, 0x0000, 0x7F62, 0xF374, 0x7D8A, 0xE707, 0x7A7D, 0xDAD8,
        0x7642, 0xCF04, 0x70E3, 0xC3A9, 0x6A6E, 0xB8E3, 0x62F2, 0xAECC,
        0x5A82, 0xA57E, 0x5134, 0x9D0E, 0x471D, 0x9592, 0x3C57, 0x8F1D,
        0x30FC, 0x89BE, 0x2528, 0x8583, 0x18F9, 0x8276, 0x0C8C, 0x809E,
        0x0000, 0x8000, 0xF374, 0x809E, 0xE707, 0x8276, 0xDAD8, 0x8583,
        0xCF04, 0x89BE, 0xC3A9, 0x8F1D, 0xB8E3, 0x9592, 0xAECC, 0x9D0E,
        0xA57D, 0xA57D, 0x9D0E, 0xAECC, 0x9592, 0xB8E3, 0x8F1D, 0xC3A9,
        0x89BE, 0xCF04, 0x8583, 0xDAD8, 0x8276, 0xE707, 0x809E, 0xF374
};

const fractcomplex twiddleFactors2[] __attribute__ ((space(auto_psv),
 aligned (fftPoints*2))) = {
        0x7FFF, 0x0000, 0x7F62, 0x0C8C, 0x7D8A, 0x18F9, 0x7A7D, 0x2528,
        0x7641, 0x30FC, 0x70E2, 0x3C57, 0x6A6D, 0x471D, 0x62F2, 0x5134,
        0x5A82, 0x5A82, 0x5134, 0x62F2, 0x471D, 0x6A6D, 0x3C57, 0x70E2,
        0x30FC, 0x7641, 0x2528, 0x7A7D, 0x18F9, 0x7D8A, 0x0C8C, 0x7F62,
        0x0001, 0x7FFF, 0xF374, 0x7F62, 0xE707, 0x7D8A, 0xDAD8, 0x7A7D,
        0xCF04, 0x7641, 0xC3A9, 0x70E2, 0xB8E3, 0x6A6D, 0xAECC, 0x62F2,
        0xA57E, 0x5A82, 0x9D0E, 0x5134, 0x9593, 0x471D, 0x8F1E, 0x3C57,
        0x89BF, 0x30FC, 0x8583, 0x2528, 0x8276, 0x18F9, 0x809E, 0x0C8C
};
```

```
// Spectrum Data
fractcomplex fftData[ fftPoints ] __attribute__ ((section (".ybss, bss, ymemory"),
aligned (fftPoints * 2 * 2)));

// Window func
fractional Win[ 64 ];

// data
unsigned int resultData;
int indata;
int outdata;
int in_num;
int out_num;
int flag;
int i;

int in_buf[fftPoints]={0};
int     in_buf2[fftPoints2]={0};
int out_buf[fftPoints]={0};
int out_buf2[fftPoints2]={0};

// Timer subroutine
void __attribute__((__interrupt__, __shadow__, no_auto_psv)) _T3Interrupt(void){
        while(!ADCON1bits.DONE) {};
        resultData = ADCBUF0;   // 12bit resolution (16kHz sampling)

        indata = (int)resultData - 2048;        // singned (range -2048 : 2047)

        IFS0bits.ADIF = 0;      // clear A/D interuppt flag
        IFS0bits.T3IF = 0;      // clear T3 interuppt flag

        in_buf[in_num] = in_buf2[in_num];
        in_buf2[in_num] = in_buf[fftPoints2 + in_num] = indata;
        in_num++;
        if(in_num & fftPoints2){
                flag = 1;
                in_num = 0;
        }

        // digital-filter processing
        outdata = (out_buf[out_num] + out_buf2[out_num])/2;
        out_buf2[out_num]= out_buf[fftPoints2 + out_num];

        outdata += 2048;                                // unsigned
        if(outdata > 4095) outdata = 4095;              // overflow limit
        if(outdata < 0) outdata = 0;                    // underflow limit
        resultData = outdata >> 2;                      // 10bits trans
        OC2RS = resultData;

        out_num++;
        if(out_num & fftPoints2){
                out_num--;
        }
}

int main(void){
```

```
        TRISB = 0x010B;         // RB1(VR3), RB2(SS), RB3(AN3), RB9(OC2)
        TRISC = 0x4000;         // RC13 LED H(RED) L(GREEN)
        TRISD = 0x0300;         // RD8 LD/RUN SW, RD9 PUSH SW
        TRISF = 0x000C;         // RF2 SDA, RF3 SCL

        ADCHS = 0x0003;         // MUXA POS CH3 NEG VREF-
        ADCON1 = 0x8044;        // ADOn, integer, TMR3 trig, auto sampling
        ADCON2 = 0x0000;        // AVDD-AVSS, no scan, per 1 time, 16BUF, MUXA
        ADCON3 = 0x0113;        // 1TAD, system clk, 10Tcy = 20Tcy/2 - 1
        ADPCFG = 0xFFF7;        // AN3 is only analog
        ADCSSL = 0x0000;        // no scan

        T3CON = 0x8000;         // TMR3 ON, prescaler 1:1, gate off, FOSC/4
        PR3 = 1843-1;

        OC2CON = 0x000E;        // PWM mode, no use FAULT, TMR3 source

        DISICNT = 0x0000;       // enable interrupt

        flag = 0;
        in_num = 0;
        out_num = 0;

        IPC1bits.T3IP = 5;
        IFS0bits.T3IF = 0;
        IEC0bits.T3IE = 1;

        HanningInit(fftPoints, Win);

        // main loop
        while(1){
                if ( flag ) {
                        flag = 0;

                        VectorMultiply(fftPoints, in_buf, in_buf, Win);
                        for(i=0; i<fftPoints; i++){
                                fftData[i].real = in_buf[i];
                                fftData[i].imag = 0;
                        }

                        IFFTComplexIP( powerOf2, fftData,
                                (fractcomplex *)twiddleFactors,
                                (int) __builtin_psvpage(&twiddleFactors[0]) );

                        IFFTComplexIP( powerOf2, fftData,
                                (fractcomplex *)twiddleFactors2,
                                (int) __builtin_psvpage(&twiddleFactors2[0]));

                        for(i=0; i<fftPoints; i++){
                                out_buf[i] = fftData[i].real*fftPoints;
                        }

                        out_num = 0;
                }
        }
}
```

付録 B　dsPIC33FJ64GP802 の UART 設定

dsPIC33FJ64GP802 で RaspberryPi と接続したときに必要な UART 設定とデータ送信部の処理を以下に記載する。まずはリマップピンの割り当てと、レジスタ設定である。

```
        RPINR18bits.U1RXR = 6;          // U1RX <- RP6(RB6)
        RPOR2bits.RP5R = 3;             // U1TX <- RP5(RB5)

        U1MODE = 0x8800;
        U1STA  = 0x0400;
        U1BRG = 20;                     // 115200bps, 39628800Hz
```

これを用いて必要な PORT の入出力を設定した。次に FFT したデータを送信用に加工する部分である。各変数については本文中で述べたので、ここでの説明は省略する。

```
        if( flag1 > 500 ){
                for( i=0; i<fftPoints2; i++ ){
                        powerSpec[i] <<= 8;
                        valL[i] = (unsigned char)( powerSpec[i] & 0x00fe);
                        valH[i] = (unsigned char)( powerSpec[i]  >> 8);
                }
        }
```

flag1 がデータを間引きする間隔の設定である。今 5ms を作るために、500 カウントとしている。次にこれらのデータを送る処理である。

```
        if( flag1 > 500 ) {
                flag1 = 0;
                while(!U1STAbits.TRMT);
                U1TXREG = 0xff;
                for( i=0; i<32; i++ ){
                        while(!U1STAbits.TRMT);
                        U1TXREG = valH[i];
                        while(!U1STAbits.TRMT);
                        U1TXREG = valL[i];
                }
                in_num = 0;
                flag = 0;
        }
```

データを送信したら、FFT 処理の計算を初期化して再び計算するようにさせている。この処理により 5ms 間隔で出力信号が無信号となる。

付録 C　Python プログラム

以下は RaspberryPi にて Python で記述したリアルタイムのグラフ表示を行うプログラムである。各処理の詳細は本文を参照。

```
import numpy as np
import wiringpi as pi
from matplotlib import pyplot as plt
```

```
# UART2 used
serial = pi.serialOpen('/dev/ttyAMA1', 115200)
print("Serial OK")

# graph array
x = np.array(range(32))
y = np.array(range(32))

while True:
        try:
                # First Data(0xff) processing
                while True:
                        while True:
                                if pi.serialDataAvail(serial) :
                                        break
                        c0 = pi.serialGetchar(serial)

                        if  c0 == 255 :
                                break

                # Data receive
                for i in range(32) :
                        while True:
                                if pi.serialDataAvail(serial) :
                                        break
                        c1 = pi.serialGetchar(serial)

                        while True:
                                if pi.serialDataAvail(serial) :
                                        break
                        c2 = pi.serialGetchar(serial)
                        c = c1 * 256 + c2
                        print(c)

                        y[i] = c

                # graph processing
                label = ["0", "", "", "", "1000", "", "", "", "2000",
                                "", "", "", "3000", "", "", "", "4000",
                                "", "", "", "5000", "", "", "", "6000",
                                "", "", "", "7000", "", "", ""]

                plt.xlim(0, 32)
                plt.xlabel("Frequency [Hz]")
                plt.ylabel("PowerSpec [a.u.]")
                plt.grid(linestyle='dotted')

                # histgram
                plt.bar(x, y, tick_label=label)
                plt.pause(0.1)

                # graph clear
                plt.gca().clear()

        # except processing
        except KeyboardInterrupt:
```

```
                pi.serialClose(serial)
                break
```

付録 D　外部 D/A コンバータ (MCP4822) の設定

本文で使用した外部 D/A コンバータの設定について記載する。詳細についてはマニュアルを参照 (http://ww1.microchip.com/downloads/en/DeviceDoc/21953a_JP.pdf：2021 年 2 月現在) のこと。まずは MCP4822 のコントロールビットの設定である。

```
// MCP4822 control bit
#define selDAC_A        0x3000
#define selDAC_B        0xb000
#define GAIN_2x         0x9000
#define GAIN_1x         0xb000
#define DAC_DOWN        0xa000
#define DAC_OE          0xb000
```

次に dsPIC33FJ64GP802 における初期のレジスタ設定である。

```
        RPINR20bits.SDI1R = 9;  // SDI1 <- RP9
        RPOR4bits.RP8R = 7;     // RP8 <- SDO1 (7)
        RPOR5bits.RP10R = 8;    // RP10 <- sck1out (8)

        SPI1CON1 = 0x067B;
        SPI1CON2 = 0x4000;
        SPI1STAT = 0xA000;

        TmpData  = SPI1BUF;                             // clear SPI receive flag
        TmpData  = selDAC_A & GAIN_1x & DAC_OE;         // set DAC control bit
        TmpData |= 2048;                                // center position

        PORTBbits.RB11 = 0;                     // MCP4822 #CS assert
        SPI1BUF = TmpData;                      // send data
        while(SPI1STATbits.SPIRBF == 0);        // waiting transmittion complete
        PORTBbits.RB11 = 1;                     // MCP4822 #CS negate
```

そして FFT 処理した信号を実際に SPI で通信を行うときの処理を以下に示す。これらはサンプリング周波数のタイマー内で実行されており、ResultData が計算結果の値である。

```
        TmpData  = SPI1BUF;                             // clear SPI receive flag
        TmpData  = selDAC_A & GAIN_1x & DAC_OE;         // set DAC control bit
        TmpData |= ResultData;                          // set value

        PORTBbits.RB11 = 0;                     // MCP4822 #CS assert
        SPI1BUF = TmpData;                      // send data
        while(SPI1STATbits.SPIRBF == 0);        // waiting transmittion complete
        PORTBbits.RB11 = 1;                     // MCP4822 #CS negate
```

付録 E　回路図

最終作品で構成した回路図を以下に示す。なお RaspberryPi4 と接続するときは UART2 なので、RaspberryPi の GPIO0(Tx) に dsPIC の Rx を、GPIO1(Rx) に dsPIC の Tx を接続した。

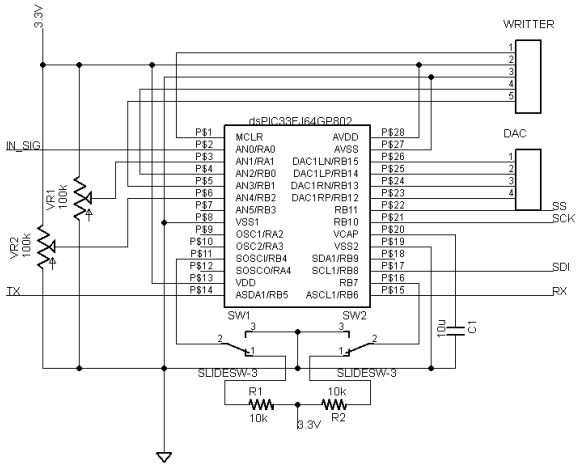

図 64　dsPIC33FJ64GP802 の周辺回路

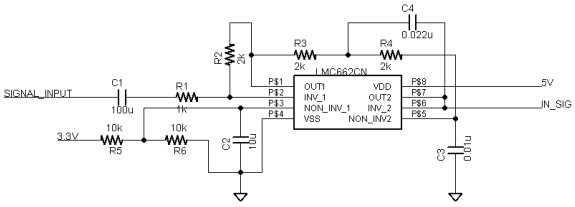

図 65　入力部フィルタ

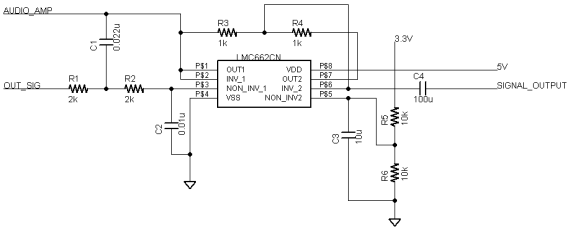

図 66　出力部フィルタ

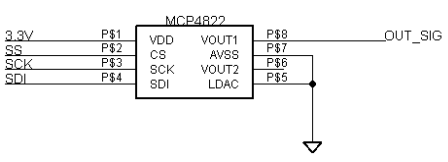

図 67　MCP4822 の接続

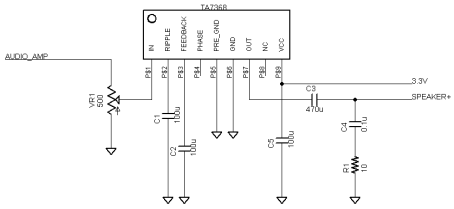

図 68　オーディオ用 IC 周辺回路

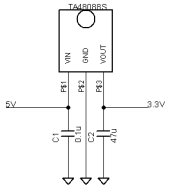

図 69　電源レギュレータ回路

付録 F　dsPIC33FJ64GP802 FFT プログラム (フィルタあり)

最終作品で実装したプログラムの全文を以下に示す。フィルタ処理は LPF と HPF で、スイッチを切り替えることによって特性を変えることができる。

```
#include <p33FJ64GP802.h>
#define FCY 39628800L
#include <libpic30.h>
#include <dsp.h>

// MCP4822 control bit
#define selDAC_A        0x3000
#define selDAC_B        0xb000
#define GAIN_2x         0x9000
#define GAIN_1x         0xb000
#define DAC_DOWN        0xa000
#define DAC_OE          0xb000

#define powerOf2                6
#define fftPoints       (1<<powerOf2) // N=64, Fs=16kHz, Fs/N=250Hz
#define fftPoints2      fftPoints/2
#define fftMask         fftPoints - 1;
#define Fs      16000

_FGS(GWRP_OFF & GCP_OFF);
_FOSCSEL(FNOSC_FRC);
_FOSC(FCKSM_CSECMD & OSCIOFNC_ON & POSCMD_NONE);
_FWDT(FWDTEN_OFF);

// twiddleFactors
const fractcomplex twiddleFactors[] __attribute__ ((space(auto_psv),
aligned (fftPoints * 2))) = {
        0x7fff, 0x0000 ,0x7f62, 0xf375 ,0x7d8a, 0xe708 ,0x7a7d, 0xdad9,
        0x7642, 0xcf05 ,0x70e3, 0xc3aa ,0x6a6e, 0xb8e4 ,0x62f2, 0xaecd,
        0x5a82, 0xa57f ,0x5134, 0x9d0f ,0x471d, 0x9593 ,0x3c57, 0x8f1e,
        0x30fc, 0x89bf ,0x2528, 0x8584 ,0x18f9, 0x8277 ,0x0c8c, 0x809f,
        0x0000, 0x8001 ,0xf375, 0x809f ,0xe708, 0x8277 ,0xdad9, 0x8584,
        0xcf05, 0x89bf ,0xc3aa, 0x8f1e ,0xb8e4, 0x9593 ,0xaecd, 0x9d0f,
        0xa57f, 0xa57f ,0x9d0f, 0xaecd ,0x9593, 0xb8e4 ,0x8f1e, 0xc3aa,
        0x89bf, 0xcf05 ,0x8584, 0xdad9 ,0x8277, 0xe708 ,0x809f, 0xf375
};

const fractcomplex twiddleFactors2[] __attribute__ ((space(auto_psv),
aligned (fftPoints*2))) = {
        0x7fff, 0x0000 ,0x7f62, 0x0c8c ,0x7d8a, 0x18f9 ,0x7a7d, 0x2528,
        0x7642, 0x30fc ,0x70e3, 0x3c57 ,0x6a6e, 0x471d ,0x62f2, 0x5134,
        0x5a82, 0x5a82 ,0x5134, 0x62f2 ,0x471d, 0x6a6e ,0x3c57, 0x70e3,
        0x30fc, 0x7642 ,0x2528, 0x7a7d ,0x18f9, 0x7d8a ,0x0c8c, 0x7f62,
        0x0000, 0x7fff ,0xf375, 0x7f62 ,0xe708, 0x7d8a ,0xdad9, 0x7a7d,
        0xcf05, 0x7642 ,0xc3aa, 0x70e3 ,0xb8e4, 0x6a6e ,0xaecd, 0x62f2,
        0xa57f, 0x5a82 ,0x9d0f, 0x5134 ,0x9593, 0x471d ,0x8f1e, 0x3c57,
        0x89bf, 0x30fc ,0x8584, 0x2528 ,0x8277, 0x18f9 ,0x809f, 0x0c8c
};
```

```
// Spectrum
fractcomplex fftData[ fftPoints ] __attribute__ ((section (".ybss, bss, ymemory"),
aligned (fftPoints * 2 * 2)));

// powerSpec
fractional      powerSpec[ fftPoints2 ];

// Window Func
fractional Win[ 64 ];

// data
unsigned int adData, adData2, adData3;
unsigned int ResultData;
unsigned int TmpData;
int indata;
unsigned int indata2;
unsigned int indata3;
int outdata;
int in_num;
int out_num;
int flag, flag1;
int i, count, j, c;
unsigned int Freq1, Freq2;
unsigned char LPF_flag, HPF_flag;

unsigned int FL, FH, Fi;

int in_buf[fftPoints]={0};
int     in_buf2[fftPoints2]={0};
int out_buf[fftPoints]={0};
int out_buf2[fftPoints2]={0};

int valH[fftPoints2] = {0};
int     valL[fftPoints2] = {0};

// Timer subroutine
void __attribute__((__interrupt__, __shadow__, no_auto_psv)) _T3Interrupt(void){
        AD1CHS0  = 0x0000;
        AD1CON1bits.SAMP = 1;
        while( AD1CON1bits.SAMP ){};
        while( !AD1CON1bits.DONE ){};
        AD1CON1bits.DONE = 0;

        adData = ADC1BUF0;
        indata = (int)adData - 2048;

        in_buf[in_num] = in_buf2[in_num];
        in_buf2[in_num] = in_buf[fftPoints2 + in_num] = indata;
        in_num++;
        if( in_num & fftPoints2 ){
                AD1CHS0  = 0x0001;
                AD1CON1bits.SAMP = 1;
                while(AD1CON1bits.SAMP){};
                while(!AD1CON1bits.DONE){};
                AD1CON1bits.DONE = 0;
```

```
                adData2 = ADC1BUF0;
                if( adData2 > 4095 )    adData2 = 4095;
                if( adData2 < 0 )       adData2 = 0;
                indata2 = adData2;

                flag = 1;
                in_num = 0;

                AD1CHS0  = 0x0004;
                AD1CON1bits.SAMP = 1;
                while( AD1CON1bits.SAMP ){};
                while( !AD1CON1bits.DONE ){};
                AD1CON1bits.DONE = 0;

                adData3 = ADC1BUF0;
                if( adData3 > 4095 )    adData3 = 4095;
                if( adData3 < 0 )       adData3 = 0;
                indata3 = adData3;
        }

        // digital-filter processing
        outdata = (out_buf[out_num] + out_buf2[out_num])/2;
        out_buf2[out_num]= out_buf[fftPoints2 + out_num];


        outdata += 2048;                        // unsigned
        if( outdata > 4095 ) outdata = 4095;    // overflow limit
        if( outdata < 0 ) outdata = 0;          // underflow limit
        ResultData = outdata;

        out_num++;
        if( out_num & fftPoints2 ){
                out_num--;
        }

        TmpData  = SPI1BUF;                             // clear SPI receive flag
        TmpData  = selDAC_A & GAIN_1x & DAC_OE;         // set DAC control bit
        TmpData |= ResultData;                          // set value

        PORTBbits.RB11 = 0;                     // MCP4822 #CS assert
        SPI1BUF = TmpData;                      // send data
        while( SPI1STATbits.SPIRBF == 0 );      // waiting transmittion complete
        PORTBbits.RB11 = 1;                     // MCP4822 #CS negate

        count++;
        IFS0bits.T3IF = 0;                      // clear T3 interuppt flag
}

int main (void){
        /* Configure Oscillator to operate the device at 40MHz.
         * Fosc= Fin*M/(N1*N2), Fcy=Fosc/2
         * Fosc= 7.37M*43/(2*2) = approx 80Mhz for 7.37M input clock
         * */

        PLLFBD=41;              /* M=43 */
        CLKDIVbits.PLLPOST=0;   /* N1=2 */
```

```
CLKDIVbits.PLLPRE=0;    /* N2=2 */
OSCTUN=0;

/*      Initiate Clock Switch to FRC with PLL (NOSC=0b001)
 *      and wait for clock switch to occur.
 *      */

__builtin_write_OSCCONH(0x01);
__builtin_write_OSCCONL(0x01);
while ( OSCCONbits.COSC != 0b01 );
while( !OSCCONbits.LOCK );

TRISA = 0x000F;         // AN0, AN1 input, RA2 - RA3 are SW
TRISB = 0x02D4;         // RB2(AN4), RB4(SW), RB6(RX), RB7(SW), RB9(SDI)
ODCB = 0x0000;          // open drain is not used

AD1CON1  = 0x84E0;      // SSRC = 111, ASAM = 0
AD1CON2  = 0x0000;
AD1CON3  = 0x010B;      //6Tcy
AD1CSSL  = 0x0000;
AD1PCFGL = 0xFFEC;      // only AN0, AN1 for analog

RPINR20bits.SDI1R = 9;  // SDI1 <- RP9
RPOR4bits.RP8R = 7;     // RP8 <- SDO1 (7)
RPOR5bits.RP10R = 8;    // RP10 <- sck1out (8)

SPI1CON1 = 0x067B;
SPI1CON2 = 0x4000;
SPI1STAT = 0xA000;

TmpData  = SPI1BUF;                             // clear SPI receive flag
TmpData  = selDAC_A & GAIN_1x & DAC_OE;         // set DAC control bit
TmpData |= 2048;                                // center position

PORTBbits.RB11 = 0;                     // MCP4822 #CS assert
SPI1BUF = TmpData;                      // send data
while( SPI1STATbits.SPIRBF == 0 );      // waiting transmittion complete
PORTBbits.RB11 = 1;                     // MCP4822 #CS negate

RPINR18bits.U1RXR = 6;  // U1RX <- RP6(RB6)
RPOR2bits.RP5R = 3;     // U1TX <- RP5(RB5)

U1MODE = 0x8800;
U1STA  = 0x0400;
U1BRG = 20;             // 115200bps, 39628800Hz

T3CON = 0x8000;         // timer3 on prescaler 1:1  nouse of gate use Fosc
PR3 = 2477-1;           // 16kHz

DISICNT = 0x0000;       // enable interrupt

in_num = 0;
out_num = 0;

IPC2bits.T3IP = 5;
IFS0bits.T3IF = 0;
```

```
        IEC0bits.T3IE = 1;

        PORTBbits.RB7 = 0;
        PORTBbits.RB4 = 0;

        flag1 = 0;
        LPF_flag = 0;
        HPF_flag = 0;

        HanningInit(fftPoints, Win);

        while (1){
                if(!PORTBbits.RB4)      HPF_flag = 1;
                if(PORTBbits.RB4)       HPF_flag = 0;
                if(!PORTBbits.RB7)      LPF_flag = 1;
                if(PORTBbits.RB7)       LPF_flag = 0;

                if ( flag ) {
                        flag = 0;
                        flag1++;

                        VectorMultiply(fftPoints, in_buf, in_buf, Win);

                        for( i=0; i<fftPoints; i++ )
                        {
                                fftData[i].real = in_buf[i];
                                fftData[i].imag = 0;
                        }

                        IFFTComplexIP( powerOf2, fftData,
                                (fractcomplex *)twiddleFactors,
                                (int) __builtin_psvpage(&twiddleFactors[0]));

                        // HPF
                        Freq1 = indata2 * 32 >> 12;
                        if ( Freq1 > 31 ) Freq1 = 31;

                        // LPF
                        Freq2 = indata3 * 32 >> 12;
                        if ( Freq2 > 31 ) Freq2 = 31;

                        if( LPF_flag && HPF_flag ){
                        }
                        if( !LPF_flag && HPF_flag ){
                                HPF_flag = 0;
                                for( i=0; i<Freq1; i++ ){
                                        fftData[i].real = fftData[i].imag = 0;
                                        fftData[63-i].real = 0;
                                        fftData[63-i].imag = 0;
                                }
                        }
                        if( LPF_flag && !HPF_flag ){
                                LPF_flag = 0;
                                for( i=Freq2; i<32; i++ ){
                                        fftData[i].real = fftData[i].imag = 0;
                                        fftData[63-i].real = 0;
```

```
                                        fftData[63-i].imag = 0;
                                }
                        }
                if( !LPF_flag && !HPF_flag );

                        SquareMagnitudeCplx( fftPoints2, fftData, powerSpec );

                        if ( flag1 > 500 ) {
                                for( i=0; i<fftPoints2; i++ ){
                                        powerSpec[i] <<= 8;
                                        valL[i] = (unsigned char)( powerSpec[i] &
                                                                        0x00fe);
                                        valH[i] = (unsigned char)( powerSpec[i]
                                                                        >> 8);
                                }
                        }

                        IFFTComplexIP( powerOf2, fftData,
                                (fractcomplex *)twiddleFactors2,
                                (int) __builtin_psvpage(&twiddleFactors2[0]) );

                        for( i=0; i<fftPoints; i++ ){
                                out_buf[i] = fftData[i].real*fftPoints;

                        }

                        out_num = 0;
                }
                if ( flag1 > 500 ) {
                    flag1 = 0;
                        while( !U1STAbits.TRMT );
                        U1TXREG = 0xff;
                        for( i=0; i<fftPoints2;i++ ){
                                while( !U1STAbits.TRMT );
                                U1TXREG = valH[i];
                                while( !U1STAbits.TRMT );
                                U1TXREG = valL[i];
                        }
                        in_num = 0;
                        flag = 0;
                }
        }
        return 0;
}
```